VŠB – TECHNICKÁ UNIVERZITA OSTRAVA

Fakulta elektrotechniky a informatiky

Katedra informatiky

Prezentace modelů metodiky BPM pomocí ontologie (OWL)

BPM Model Representation by Ontology

2012 Tereza Moudrá, mou038

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Tereza Moudrá**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Reprezentace modelů metodiky BPM pomocí ontologie (OWL)
BPM Model Representation by Ontology

Zásady pro vypracování:

Výstupem práce je zajistit ukládání modelů vytvořených v nástroji BPStudio do souborů formátu OWL, které slouží k uložení ontologie. Podporované bude i načítání takto vytvořených OWL souborů zpět do BPStudia.
Doporučený postup:
1. Vytvořte ontologii metodiky BPM.
2. Nadefinujte mapování modelu BPStudia do ontologie, která je rozšířením ontologie metodiky BPM.
3. Naprogramujte ukládání/ načtení modelu do/z ontologií.

Seznam doporučené odborné literatury:

OWL Web Ontology Language Reference [online]. Available from World Wide Web:<http://www.w3.org/TR/2004/REC-owl-ref-20040210/>.
LACY, L.W. Owl: Representing Information Using the Web Ontology Language. Trafford Publishing, 2005. ISBN 1412034485.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Jan Kožusznik, Ph.D.**

Datum zadání: 18.11.2011
Datum odevzdání: 04.05.2012

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA Fakulta elektrotechniky a informatiky OSTRAVA

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

„Prohlašuji, že jsem tuto diplomovou práci vypracovala samostatně.

Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala."

V Ostravě 3.5.2012 Podpis

Poděkování

Na tomto místě bych chtěla poděkovat vedoucímu své diplomové práce panu Ing. Kožusznikovi, Ph.D. za rady a připomínky k obsahu a formě zpracování.

Abstrakt

Tato diplomová práce se zabývá uplatněním ontologií v oboru návrhu podnikového procesu. Využívá nástroj BP Studio, pomocí něhož lze vytvářet a editovat business procesy. Součástí této práce je studium mapování jednotlivých grafických prvků BP Studia na ontologické třídy a jedince a následné vytvoření ontologie reprezentující diagram navržený právě v BP Studiu. Výsledkem této práce je implementace modulu pro vytváření ontologií z namodelovaného diagramu a také pro zpětné načítání ontologie do nástroje.

Klíčová slova

Ontologie, OWL, BPM, BP Studio, business proces, návrhový vzor Builder

Abstract

This thesis deals with the application of ontologies in the discipline of design business process. It is used tool BP Studio, which allows creating and editing business processes. Part of this work is to study the mapping of individual graphic elements from BP Studio to the ontological classes and individuals and subsequently creation of ontology representing diagram designed in BP Studio. The result of this work is to implementation a module for creating ontologies from designed diagram, and also for loading back into the ontology tool.

Keywords

Ontology, OWL, BPM, BP Studio, business process, design pattern Builder

Seznam použitých symbolů a zkratek

OWL	- Web Ontology Language
BPM	- Business Process Management
RDF	- Resource Description Framework

Seznam obrázků

Obsah

1. Úvod

Již od 80. let se začíná rozvíjet ontologické inženýrství, což je nová oblast informatiky, zabývající se výměnou informací mezi lidmi a stroji. Současně také vnáší do informací sémantický význam. Tato diplomová práce se zabývá použitím ontologií pro účely popsání, charakterizování a zachycení modelovaných business procesů pomocí nástroje BP Studio. Navržená ontologie by měla splňovat podmínky, podle kterých by měla být dále využitelná v mnoha počítačových zpracování. Součástí této diplomové práce je také implementační část, která by měla zajišťovat převod modelu z BP Studia na ontologie a zase zpět. V současné chvíli nelze s modely BP Studia pracovat jinak, než právě pomocí tohoto nástroje. Díky propojení mezi BP Studiem a ontologiemi by bylo možné manipulovat s modely také pomocí nástrojů pro práci s ontologiemi a XML soubory a umožňovalo by tak jejich další transformaci a integraci s jinými modely.

Druhá kapitola popisuje teorii ontologií. Popisuje jednotlivé prvky ontologie a jejich použití, dále také význam ontologií, jejich účel a v neposlední řadě jazyky, pomocí kterých se ontologie dají vytvářet.

Třetí kapitola umožňuje náhled do prostředí nástroje Protégé, který slouží k pohodlnému modelování ontologií. Čtvrtá kapitola seznamuje s použitou OWL knihovnou, pomocí které je implementační část této diplomové práce naprogramována. V kapitole jsou uvedeny konkrétní příklady použití využitých metod.

V páté kapitole je představen modelovací nástroj BP Studio a také popsán příklad modelu business procesu, na kterém je demonstrováno převedení diagramu do ontologie. Šestá kapitola se již zabývá samotným převáděním prvků z BP Studia do ontologického modelu. Je zde popsán kompletní proces převodu.
V sedmé a osmé kapitole je popsána implementační část diplomové práce. Tyto kapitoly vysvětlují, jak lze daná data získat a jak se s nimi posléze pracuje.

2. Ontologie

Reprezentace znalostí je pojem úzce spjatý s umělou inteligencí. Od 80. letech se začal měnit důraz z hledání univerzálního algoritmu k práci se specializovanými znalostmi určité oblasti. Jinými slovy se kladl důraz na to, co má být znalostní aplikací řešeno a až poté, jak to má být řešeno. Tuto znalost využívají například expertní systémy. Při vytváření znalostního systému je důležité zvolit vhodný způsob reprezentace znalostí. Tomuto způsobu reprezentace se často říká reprezentační schéma. Jedná se o soubor pravidel a postupů, které je nutné dodržet pro zachycení daných znalostí.

2.1. Historie

Pojem ontologie se vyskytoval již v době starověkého Řecka v oboru filosofie. Většího významu tento pojem dosáhl v 18. století v metafyzice, kdy představoval podstatu bytí. Avšak v 80. letech se pojem ontologie začal využívat i v oblasti získávání, reprezentace a sdílení znalostí, v oboru znalostního inženýrství. Ze znalostního inženýrství, se později zrodil nový obor, a to ontologické inženýrství. Ontologické inženýrství zahrnuje soubor aktivit, které se týkají procesu vývoje, životního cyklu a metod konstrukce ontologií. Mezi jednotlivé aktivity patří například analýza, zda lze daná ontologie vytvořit, konstrukce ontologického modelu, integrace a rozšíření existujícího řešení, vytvoření formálního modelu, přizpůsobení ontologie novým požadavkům a využití v praxi [1].

2.2. Význam ontologie

Ontologie je pojem označující domluvenou terminologii pro určitou aplikační oblast, která umožňuje sdílení znalostí z této oblasti. Ontologie umožňuje vytváření informací o informacích. Tato metadata právě vytvářejí významové informace. V informatice se ontologie přirovnává k taxonomiím, jenž zachycují hierarchii pojmů, vyjadřují vztahy mezi koncepty naší reality ve smyslu zobrazení nadtříd a podtříd. Z definic významných informatiků vychází, že je ontologie formální konceptualizací. To znamená, že pro svou tvorbu využívá jazyky s přesně definovanou syntaxí. Navíc znalosti, které ontologie poskytují, jsou přístupné i jiným a ontologie by měly být sdílené.

Význam ontologie v informačních oborech je svým způsobem stejný jako v oblasti filosofie. Oba tyto obory se snaží pomocí ontologie zachytit koncepty našeho světa

a charakterizovat je. Cílem ontologie není nic jiného, než se dorozumět mezi lidmi navzájem. V počítačovém světě je také cílem ontologie propojit komunikaci mezi lidmi a stroji a také mezi stroji navzájem. Dalším významným cílem ontologie je například znovupoužití doménových znalostí.

2.3. Vysvětlení pojmu

Pojem ontologie popisuje to, co existuje a může být reprezentováno v informačním systému. Ve většině literaturách zaměřených na ontologické inženýrství se uvádí mnoho různých definic tohoto pojmu. Nejčastěji je však ontologie vyjádřena definicí T. Grubera [2], kterou později ještě W. Borst [3] upravil do její finální podoby: "*Ontologie je formální, explicitní specifikace sdílené konceptualizace*".

Podle této definice by ontologie měla splňovat určitá pravidla. Formálnost znamená, že ontologie musí být vytvořena jazykem s přesně definovanou syntaxí (popřípadě i sémantikou). Explicitní ontologie je taková, která zpřístupňuje své informace i jiným. Pojem sdílenosti udává, aby ontologie nebyla pouze individuální záležitosti, ale výsledkem práce zájmové skupiny lidí. Nejdůležitějším bodem definice je však pojem konceptualizace. Tento pojem popisuje, co vlastně ontologie je. Což znamená, že se jedná o systém pojmů, které modelují realitu. Vlastnosti sdílenost a formálnost ontologie nemusí být vždy dodržovány. Existují případy kdy je ontologie neformální či semi-formální nebo neprošla kolektivní diskuzí. Mezi další vlastnost ontologie patří, že jsou znovu použitelné (reusable) pro různé aplikace.

2.4. Účel ontologií

Mezi základní způsoby využití ontologií se bezesporu řadí podpora porozumění mezi lidmi, podpora komunikace mezi počítačovými systémy či usnadnění návrhu znalostně orientovaných aplikací. Jako praktickou aplikaci ontologií lze jmenovat například znalostní management ve firmách, kde daná ontologie napomáhá efektivnímu fungování organizace, usnadňuje vyhledávání informací a zabezpečuje jejich konzistenci. Dalším příkladem použití ontologie je zpracování přirozeného jazyka, vyhledávání informací nebo tvorba webových portálů.

2.5. Kategorie ontologie

Ontologii je možné členit podle oborových oblastí [4].

1. Terminologické (lexikální) ontologie jsou velmi podobné pokročilým tezaurům a využívají se tedy zejména v knihovnictví a dalších oborech zaměřených na zpracování textu. Již podle názvu je zřejmé, že jejich důležitým rysem je důraz na termíny.
2. Informační ontologie je nadstavba databázových konceptuálních schémat.
3. Znalostní ontologie rozvíjejí oblast reprezentace znalostí.

Další dělení lze uplatnit podle předmětu formalizace. Základními typy jsou:

1. Doménové ontologie jsou nejčastěji používaným typem. Jejich obsahem je určitá specifická oblast, tzv. doména, která ohraničuje informace, kterých se ontologie týká. Příkladem může být například ontologie zaměřená na lékařskou oblast (UMLS) nebo strukturu a činnost podniku (Enterprise Ontology).
2. Generické ontologie zachycují obecné zákonitosti jako je například zachycení času, vzájemné pozice objektů či složení objektů.
3. Úlohové ontologie se zaměřují na procesy odvozování, nikoli na zachycení znalostí o světě. Jedná se o modely řešení problémů (diagnostika, konfigurace, plánování).
4. Aplikační ontologie je model převzatý a přizpůsobený pro konkrétní aplikaci, který zahrnuje doménovou i úlohovou část.

2.6. Struktura ontologií

Třída (Class)

Základními prvky ontologie jsou třídy, které označují množiny konkrétních objektů se stejnými vlastnostmi. V některých případech se třída označuje i termínem koncept, kategorie nebo rámec. Třídy neobsahují metody jako je tomu v objektově-orientovaných modelech. Třídy jsou seskupovány do hierarchie, což znamená, že mohou existovat nadtřídy i podtřídy. Vytvořením nadtříd a podtříd se vytváří struktura ontologie. Třídy, které mají podmínky nutnosti i postačitelnosti se označují jako definované. Ty které tyto podmínky definovány nemají se označují jako primitivní.

Na obrázku 1 je znázorněna grafická značka třídy, která se bude v této diplomové práci využívat pro její znázornění. Tato třída reprezentuje množinu `Zaměstnání`.

Obrázek 1: Grafická značka třídy

Jedinec (Individual)

Tento prvek ontologie lze také nazývat individuum nebo instance. Představuje konkrétní objekt reálného světa a náleží tak do některé z definovaných tříd. Jinými slovy jej lze označit jako instanci třídy. Je také možné, aby jedinec patřil i do více tříd najednou. Existuje však rozdíl mezi instancí a jedincem. Instance vždy patří k některé třídě, ale jedinec může být do ontologie přidán bez vazby na třídu. Primární účel ontologie je ale popis konceptu, tedy tříd, nikoli konkrétních objektů.

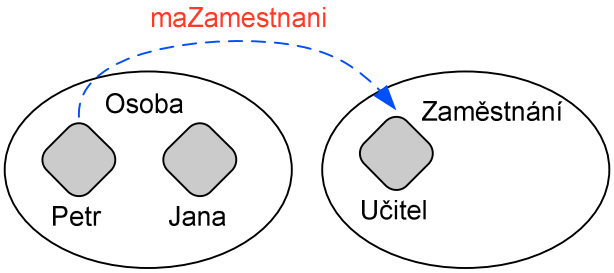
Občas může být zařazení entity nejednoznačné. V některých případech je vhodné entitu prohlásit za třídu a někdy zase za jedince. Ukázkovým příkladem může být zařazování živočišných druhů. V ontologii, která popisuje chování zvířat, je vhodné jednotlivé živočišné druhy prezentovat pomocí tříd. Oproti tomu v ontologii, která popisuje vývoj organismu, je zřejmé, že půjde spíše o individua, jelikož jednotlivá zvířata nemá smysl uvažovat. Na obrázku 2 je znázorněna grafická podoba jedince. Tento jedinec představuje konkrétní zaměstnání `Učitel`. Je tedy možné jej zařadit do třídy `Zaměstnání`, jako je tomu na pravém obrázku.

Obrázek 2: Grafická značka jedince

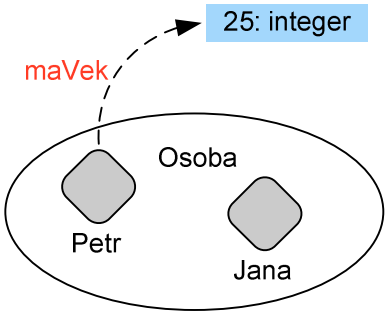
Vlastnost (Property)

Dalším důležitým prvkem struktury ontologie je bezesporu vztah mezi jedinci. Tento vztah se nazývá vlastnost nebo v jiné terminologii také relace. Vlastnost lze specifikovat pomocí logických podmínek (globální nebo lokální omezení). Vlastnosti nejsou napevno spojeny s žádnou třídou jako část její definice, ale součástí vlastností je i definiční obor (domain) a obor hodnot (range), kterými lze vlastnost omezit a definovat, které třídy resp. jedinci tříd mohou vlastnost využít. Stejně tak jako třídy i vlastnosti lze hierarchizovat.

Existuje několik typů vlastností. Prvním z nich je, jak už bylo dříve zmíněno, objektová vlastnost (object property), která reprezentuje vztah mezi objekty - jedinci. Tento druh spojení však není jediný. Jedinci mohou být spojeni i s primitivními hodnotami, které neodpovídají žádnému objektu. Takový vztah se označuje jako datová vlastnost (data property). Obor hodnot této vlastnosti bývá vymezen některým ze základních datových typů, jako je string, integer, float, enum, apod. Na obrázku 3 je znázorněna objektová vlastnost, která spojuje jedince z různých tříd a udává tak jejich souvislosti. Je tedy možné definovat vztah mezi zaměstnáním a člověkem a říci tak, že konkrétní osoba `Petr` pracuje jako `Učitel`. Obrázek 4 pomocí datové vlastnosti zobrazuje fakt, že osoba `Petr` má 25 let.



Obrázek 3: Grafická značka objektové vlastnosti



Obrázek 4: Grafická značka datové vlastnosti

Axiomy

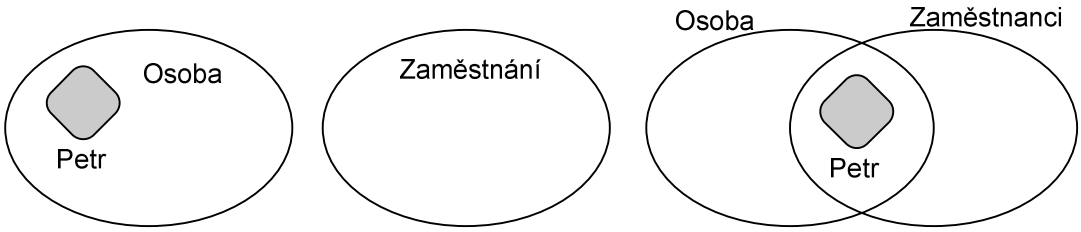
Kromě definování oboru hodnot a definičního oboru lze také určit ekvivalenci tříd a vlastností, disjunktnost tříd nebo třeba rozdělení třídy na podtřídy. V jazyce DAML+OIL jsou axiomy součástí definice tříd a vlastností.

Disjunktnost tříd

Tato vlastnost se použije tehdy, jestliže je potřeba jednotlivé třídy od sebe oddělit a odlišit. Jinými slovy neexistuje jedinec, který by patřil oběma třídám. Musí patřit pouze jedné z nich. Samozřejmě

však záleží na úhlu pohledu, ze kterého se daná ontologie modeluje. V jednom případě je vhodné udělit třídám disjunktnost, v jiném zase ne.

Například třídy `Osoba` a `Zaměstnání` jsou disjunktní, jelikož nelze říci, že jedinec `Petr` by byl také jedincem třídy `Zaměstnání`. Oproti tomu pokud by existovaly třídy `Osoba` a `Zaměstnanci`, bylo by možné říct, že jedinec `Petr` je součástí třídy `Osoba` i `Zaměstnanci`, jelikož má zaměstnání.

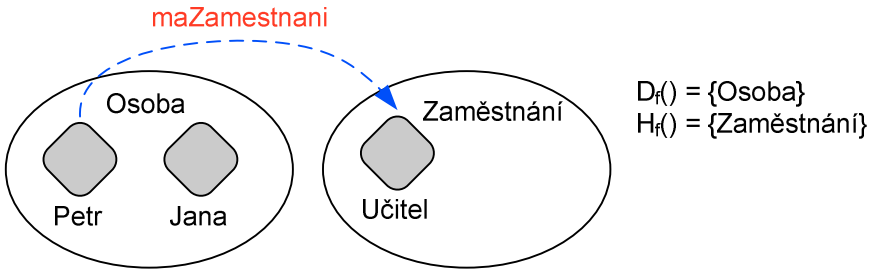


Obrázek 5: Disjunktnost tříd

Definiční obor a obor hodnot

Tyto dva parametry se definují u objektových vlastností. Definiční obor i obor hodnot uvádí mezi kterými jedinci, z kterých tříd může vlastnost existovat. Díky těmto specifikacím lze zabránit případným nekonzistencím ontologie. Definiční obor udává, u které třídy se může vlastnost objevit. Obor hodnot definuje, jakou hodnotu může vlastnost nabývat. Stanovení definičního oboru a oboru hodnot se také nazývá globální omezení vlastnosti.

Například lze omezit vlastnost `maZamestnani`, aby byla použitelná pouze pro třídy `Osoba` a `Zaměstnání`. Jako definiční obor se v tomto případě nastaví třída `Osoba` a obor hodnot třída `Zaměstnání`.

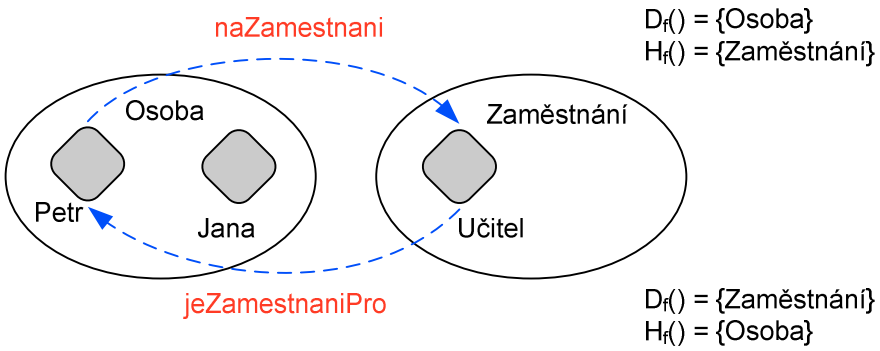


Obrázek 6: Definiční obor a obor hodnot

Inverzní vlastnost

Jedná se o objektovou vlastnost, která je definována k již vytvořené vlastnosti. Má pouze prohozen definiční obor s oborem hodnot. Inverzí vlastnost spolu s původní vlastností zajišťují obousměrný vztah mezi jedinci daných tříd.

Příkladem může být inverzní vlastnost k vlastnosti `maZamestnani`. Stejně tak je vhodné totiž definovat vztah od zaměstnání `Učitel` ke konkrétní osobě. V tomto případě je možné vlastnost pojmenovat `jeZamestnaniPro`.



Obrázek 7: Inverzní vlastnost

Lokální omezení vlastnosti

Týká se zejména sémantiky třídy. Do této sekce spadá hned několik druhů omezení: kvantitativní omezení (existenční, univerzální), kardinální omezení (<, >, =) a hasValue omezení. Existenční omezení (some) udává, že každý jedinec třídy má danou vlastnost. Univerzální omezení (only, no values except, onlyValuesFrom) definuje, že jedince může mít pouze danou vlastnost a žádnou jinou. Kardinální omezení udává počet vztahů, kterých se jedinec, náležející určité třídě, má zúčastnit. HasValue omezení vytváří vazbu mezi skupinou všech jedinců vybrané třídy a jedním konkrétním jedincem a zároveň připouští vztah s jiným jedincem z jiné třídy.

Axiom uzávěru vlastnosti

Toto omezení přesně specifikuje obor hodnot pro určitou vlastnost. Je tvořeno sjednocením všech existenciálních omezení, což znamená, že vlastnost může působit pouze na definované třídy a na žádné jiné. Axiom uzávěru tedy obsahuje jako omezení some tak i only.

2.7. Jazyky

Existuje celá řada jazyků, které dokáží zachytit podstatu ontologií. V následující odstavcích jsou zmíněny pouze jazyky novější doby, tzv. webové [5].

RDFS - RDF Schema

Jedná se o první sémantický jazyk orientovaný na RDF. RDF je určen zejména pro reprezentaci struktur webových metadat. Myšlenka RDF se zakládá na tvrzení, že každý

webových zdroj má vlastnosti, které jsou pomocí RDF charakterizovány pomocí trojice hodnot složených z podmětu, vlastnosti a předmětu.

RDFS doplňuje do RDF hlavní konstrukce z objektových systémů, což znamená třídy, vlastnosti s definičním oborem a oborem hodnot a dále také možnost hierarchie tříd i vlastností. Nelze však detailněji specifikovat podmínky příslušnosti ke třídám a neobsahuje datové typy.

DAML+OIL

Jazyk DAML+OIL je, jak už je z názvu patrné, složenina dvou jazyků, a to: DAML-ONT a OIL. DAML-ONT vycházel z předchozích studií ontologických jazyků a nově zahrnul řadu konstruktů pro vymezení vztahů tříd a hodnot vlastností. Jazyk OIL vznikl na potřebě konstrukce složitějších podmínek. Jeho podstatou je deskripční logika.

Základem DAML+OIL jsou třídy reprezentované svým jménem (URI) nebo logickým výrazem. Dále také obsahuje logické výrazy vymezující třídy, které se nazývají konstruktory. Konstruktory lze libovolně kombinovat a vytvářet tak složitější výrazy. Dalšími pojmy jsou axiomy, které definují vlastnosti tříd a vlastností, a dále také ekvivalence či disjunktnost.

OWL

Po zhruba dvou letech používání jazyku DAML+OIL vzniká jazyk OWL pod hlavičkou W3C. OWL jazyk lze rozdělit do tří verzí. Nejjednodušší z nich je OWL Lite, která disponuje pouze základními elementy a je tedy vhodná pro tvorbu méně strukturované ontologie s jednoduchými omezeními. Druhou verzí je OWL-DL. Je s ní možné realizovat například ontologickou hierarchii či kontrolu konzistence. Poslední verzí je OWL-Full, která obsahuje všechny elementy a konstrukce jazyka.

OWL dokument obsahuje hlavičku a následně definici tříd, vlastností a individuí. Hlavička je složena z důležitých informací o ontologii. Dále také může obsahovat informace o verzi dokumentu nebo informaci o vnořených ontologiích. Následující příklady popisují použití syntaxe OWL/XML. Začátek dokumentu je uvozen hlavičkou typickou pro XML soubory. Dále je uvedeno tělo ontologie pomocí elementu `Ontology`.

```
<?xml version="1.0"?>
<Ontology xmlns="http://www.w3.org/2002/07/owl#"
   xml:base="http://ontologie.cz/2011/4/Ontology1304689076651.owl"
   ontologyIRI="http://ontologie.cz/2011/4/Ontology1304689076651.owl">
   <Prefix name="owl" IRI="http://www.w3.org/2002/07/owl#"/>
</Ontology>
```

V těle ontologie dále následuje definice tříd. Třídy jsou jednoznačně popsány svým názvem - URI. Třídu lze definovat pomocí identifikátoru, výčtu prvků, které tvoří instanci třídy, omezením vlastností, sjednocením či průnikem dvou a více definic tříd nebo doplňkem definice třídy. Definice tříd, ale také i definice vlastností je popsána pomocí elementu `Declaration`. Konkrétním příkladem může být definice třídy `Process`, která je podtřídou hlavní třídy `Thing`.

```
<Declaration>
   <Class IRI="#Process"/>
</Declaration>
<SubClassOf>
   <Class IRI="#Process"/>
   <Class abbreviatedIRI=":Thing"/>
</SubClassOf>
```

Třídy lze dále charakterizovat pomocí disjunkce.

```
<DisjointClasses>
   <Class IRI="#Active"/>
   <Class IRI="#Passive"/>
</DisjointClasses>
```

Další část OWL dokumentu je věnována definici vlastností ontologie. Objektové vlastnosti je možné definovat pomocí elementu `ObjectProperty` a datové vlastnosti pomocí elementu `DatatypeProperty`. Příkladem může být definování objektové vlastnosti `hasConnectionFrom`, která má omezení v podobě definičního oboru a oboru hodnot a může tak spojovat pouze instance třídy `Process`. Dále je v příkladu uvedena definice datové vlastnosti `hasName`, která může být použita pouze pro třídu `Process` a oborem hodnot je znakový řetězec.

```
<Declaration>
   <ObjectProperty IRI="#hasConnectionFrom"/>
</Declaration>
<ObjectPropertyDomain>
   <ObjectProperty IRI="#hasConnectionFrom"/>
   <Class IRI="#Process"/>
</ObjectPropertyDomain>
<ObjectPropertyRange>
   <ObjectProperty IRI="#hasConnectionFrom"/>
   <Class IRI="#Process"/>
</ObjectPropertyRange>

<Declaration>
   <DataProperty IRI="#hasName"/>
</Declaration>
<DataPropertyDomain>
   <DataProperty IRI="#hasName"/>
   <Class IRI="#Process"/>
</DataPropertyDomain>
<DataPropertyRange>
```

```
   <DataProperty IRI="#hasName"/>
   <Datatype abbreviatedIRI="xsd:string"/>
</DataPropertyRange>
```

Do OWL dokumentu lze také samozřejmě zapsat i konkrétní jedince ontologie. V uvedeném příkladu je vytvořen jedinec `Order`, který patří do třídy `Process`. Tento jedinec má datovou vlastnost `hasDescription`, která popisuje daný proces. K vytvoření jedince a jeho přiřazení ke konkrétní třídě slouží element `ClassAssertion`. Pro definování datové vlastnosti jedince je určen element `DataPropertyAssertion`, u kterého se označuje datová vlastnost, jedinec, kterého se přiřazení týká a samotná hodnota vlastnosti.

```
<ClassAssertion>
   <Class IRI="#Process"/>
   <NamedIndividual IRI="Order"/>
</ClassAssertion>

<DataPropertyAssertion>
   <DataProperty IRI="#hasDescription"/>
   <NamedIndividual IRI="Order"/>
   <Literal datatypeIRI="&xsd;string">Description of process</Literal>
</DataPropertyAssertion>
```

3. Protégé

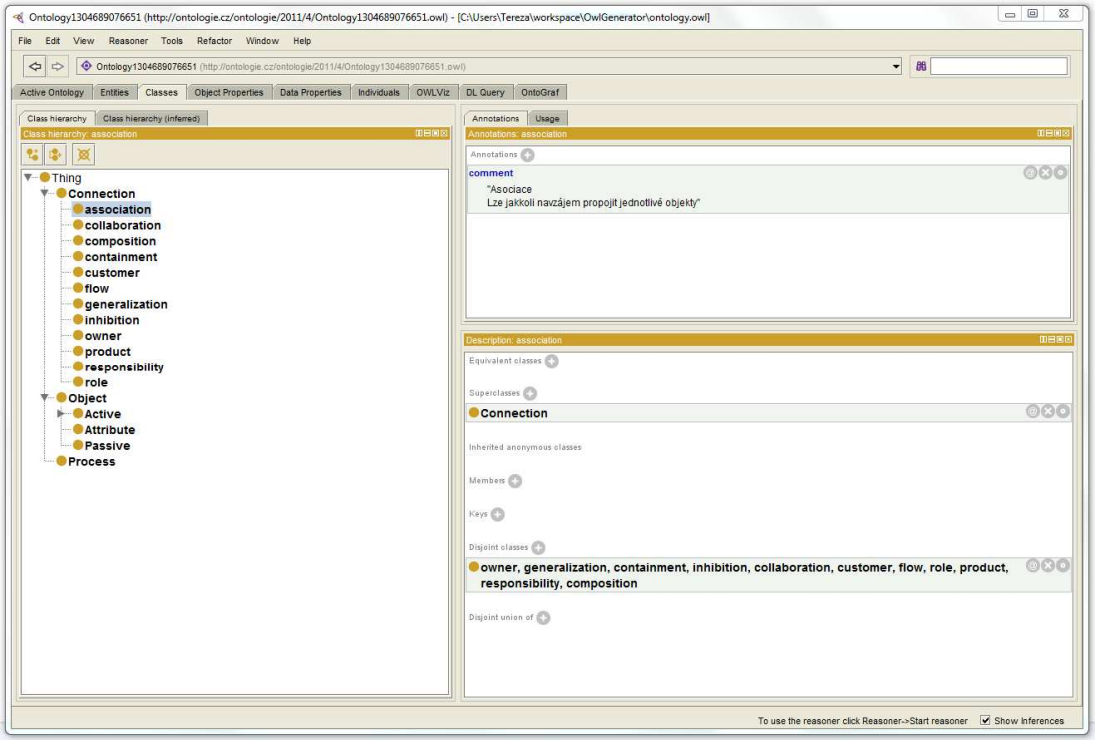
Protégé je jedním z nejrozšířenějších nástrojů pro tvorbu a správu ontologií. Je také open-source a je volně dostupný na internetu [6]. Protégé je implementován v programovacím jazyce Java a je nezávislý na platformě. Aktuální verzí Protégé je verze 4.2. Existuje také řada přídavných pluginů, které lze implementovat do stávajícího nástroje Protégé, například pro vizualizaci ontologií. Pomocí tohoto nástroje (verze 4.1) je také namodelována ontologie pro tuto diplomovou práci.

3.1. Vlastnosti nástroje

Mezi přední vlastnosti nástroje Protégé patří již zmiňovaná nezávislost na platformě. Dále také možnost ukládat ontologie v různých formátech (XML, RDF, OWL a jiné). Zároveň také dokáže pracovat s ontologiemi ve formátu (HTML, OWL, RDF, Turtle, N-Triple či N3). Další vlastností je možnost integrace s jinými aplikacemi.

3.2. Terminologie

Protégé využívá obdobnou terminologii jako ontologie, což tedy znamená, že se zde vyskytují třídy, jedinci a vlastnosti. Třída (class) představuje množinu objektů, jedinec (individual) je instancí třídy a vlastnost (property) představuje vztah mezi jedinci nebo udává hodnotu jedince. Více informaci o jednotlivých prvcích ontologie viz. kapitola 2. Ontologie.



Obrázek 8: Grafické uživatelské rozhraní Protégé

4. OWL API

OWL API je standardizovaná Java knihovna pod záštitou University of Manchester [7], která slouží pro vytváření a manipulaci s ontologiemi. Nejnovější verze je zaměřena na OWL 2. OWL API obsahuje parser a metody pro zapisování dokumentů formátů RDF/XML, OWL/XML, OWL Functional Syntax, Turtle a KRSS.

Práce s ontologií probíhá pomocí manageru OWLOntologyManager. Prostřednictvím tohoto manageru je možné s ontologií manipulovat - vytvářet, načítat již vytvořenou ontologii, ukládat či mazat. Dalším pomocným konstruktem je OWLDataFactory, pomocí něhož lze pracovat s jednotlivými prvky ontologie, jako jsou třídy, vlastnosti a jedinci.

V následujících odstavcích jsou uvedeny pouze ukázky některých funkcí knihovny, které byly později využity i v této diplomové práci. Nejsou zde uvedeny příklady demonstrující vytváření jednotlivých tříd ani datových či objektových vlastností, jelikož součástí této diplomové práce je pouze načtení již vymodelované ontologie a následné vytváření jedinců této ontologie.

Načtení již vytvořené ontologie probíhá právě pomocí manageru. Ontologie je jednoznačně určena pomocí IRI.

```
OWLOntologyManager manager = OWLManager.createOWLOntologyManager();
IRI iri = IRI.create("http://www.co-ode.org/ontologies/bmp/ontology.owl");
OWLOntology bmpOntology = manager.loadOntologyFromOntologyDocument(iri);
```

Ontologii lze načíst samozřejmě také z lokálního umístění v počítači.

```
File file = new File("/tmp/ontology.owl");
OWLOntology bmpOntology = manager.loadOntologyFromOntologyDocument(file);
```

Uložení ontologie je možné následujícím způsobem. V prvním případě se ontologie uloží ve stejném formátu, ve kterém byla načtena. Jestliže je potřeba ukládat ontologii v jiném formátu, postačí použít druhou variantu.

```
// Same format
File local = new File("/tmp/local.owl");
manager.saveOntology(bmpOntology, IRI.create(local.toURI()));

// Different format
OWLOntologyFormat format = manager.getOntologyFormat(bmpOntology);
OWLXMLOntologyFormat owlxmlFormat = new OWLXMLOntologyFormat();
if(format.isPrefixOWLOntologyFormat()) {
   owlxmlFormat.copyPrefixesFrom(format.asPrefixOWLOntologyFormat());
}
manager.saveOntology(bmpOntology,owlxmlFormat,IRI.create(file.toURI()));
```

Po načtení ontologie je také potřeba načíst jednotlivé prvky ontologie. V následujícím příkladě je uvedeno načtení množiny tříd, množiny datových a objektových vlastností a množiny jedinců.

```
Set<OWLClass> classes = bmpOntology.getClassesInSignature()
Set<OWLDataProperty> dataProperties =
bmpOntology.getDataPropertiesInSignature()
Set<OWLObjectProperty> objectProperties =
bmpOntology.getObjectPropertiesInSignature()
Set<OWLNamedIndividual> individuals =
bmpOntology.getIndividualsInSignature()
```

Součástí OWL knihovny jsou pochopitelně také metody pro vytváření jedinců ontologie, jejich přiřazení ke třídám a také pro definování datových a objektových vlastností daných jedinců. Jedinec je reprezentován pomocí třídy `OWLIndividual` a jeho přiřazení ke třídě znázorňuje třída `OWLClassAssertionAxiom`. Proměnná `pm` typu `PrefixManager` obsahuje tzv. prefix manageru, ve kterém je uvedena adresa uložení ontologie. Nejprve je potřeba načíst do proměnné `cl` třídu, do které má vytvářený jedinec patřit. Poté je pomocí metody `addAxiom` uložen do ontologie i s jeho třídním zařazením.

```
OWLClass cl = dataFactory.getOWLClass(":Process", pm);
OWLIndividual individual =
dataFactory.getOWLNamedIndividual(":Individual01", pm)
OWLClassAssertionAxiom classAssertion =
dataFactory.getOWLClassAssertionAxiom(cl, individual)
manager.addAxiom(bpmOntology, classAssertion);
```

U vytvořeného jedince lze nyní definovat jeho další atributy, jako jsou datové a objektové vlastnosti. Datová vlastnost je reprezentována pomocí třídy `OWLDataProperty` a její přiřazení k jedinci se provádí pomocí `OWLDaraPropertyAssertionAxiom`. Následující kód nastavuje jedinci jeho jméno prostřednictvím datové vlastnosti `hasName` a návaznost na ostatní jedince pomocí objektové vlastnosti `hasConnectionWith`.

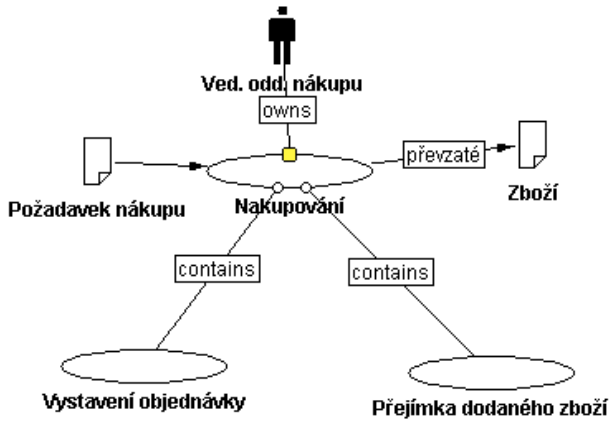
```
OWLDataProperty dataProp = dataFactory.getOWLDataProperty(":hasName", pm);
OWLDataPropertyAssertionAxiom dataAssertion =
dataFactory.getOWLDataPropertyAssertionAxiom(dataProp,individual,"Name");
manager.addAxiom(bpmOntology, dataAssertion);

OWLObjectProperty objectProp =
dataFactory.getOWLObjectProperty(":hasConnectionWith", pm);
OWLDataObjectAssertionAxiom objectAssertion =
dataFactory.getOWLDataObjectAssertionAxiom(objectProp,individual,individual2);
manager.addAxiom(bpmOntology, objectAssertion);
```

5. BP STUDIO

BP Studio je vizuální modelovací nástroj, pomocí něhož lze vytvářet modely business procesů. Budování modelů v tomto nástroji nezávisí na typu procesu ani na doméně oblasti, do které daný proces patří (průmysl, bankovní sektor, obranné systémy, služby, atd.). Model podnikového procesu je možné modelovat ze tří různých pohledů [8].

5.1. Funkční pohled

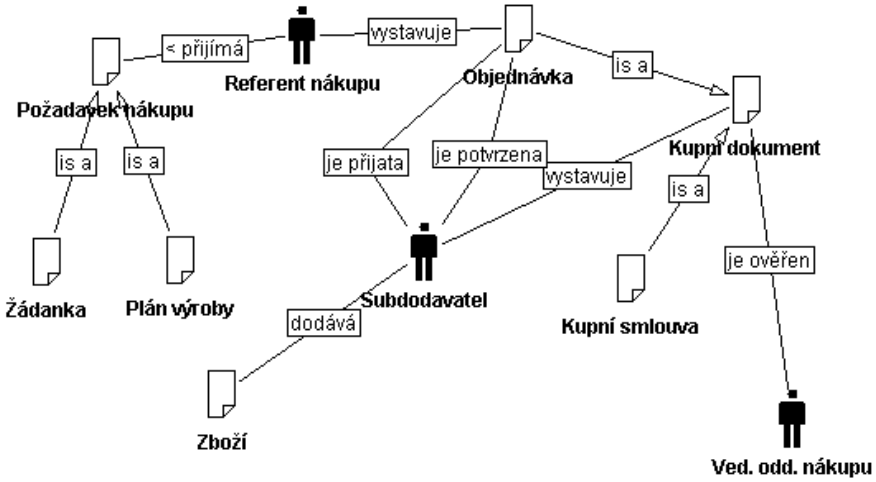
Na této úrovni modelu lze identifikovat architekturu podnikového procesu. Vykresluje tedy veškeré procesy a jejich strukturu. V tomto pohledu se lze setkat se dvěma typy vztahů mezi procesy. Těmi jsou obsažení a spolupráce. Obsažení slouží k identifikaci podprocesů. Oproti tomu vztah spolupráce definuje možnost souběžné existence dvou či více procesů. Funkční pohled vykresluje odpovědi na otázku co.



Obrázek 9: Diagram modelovaný ve funkčním pohledu

5.2. Objektový pohled

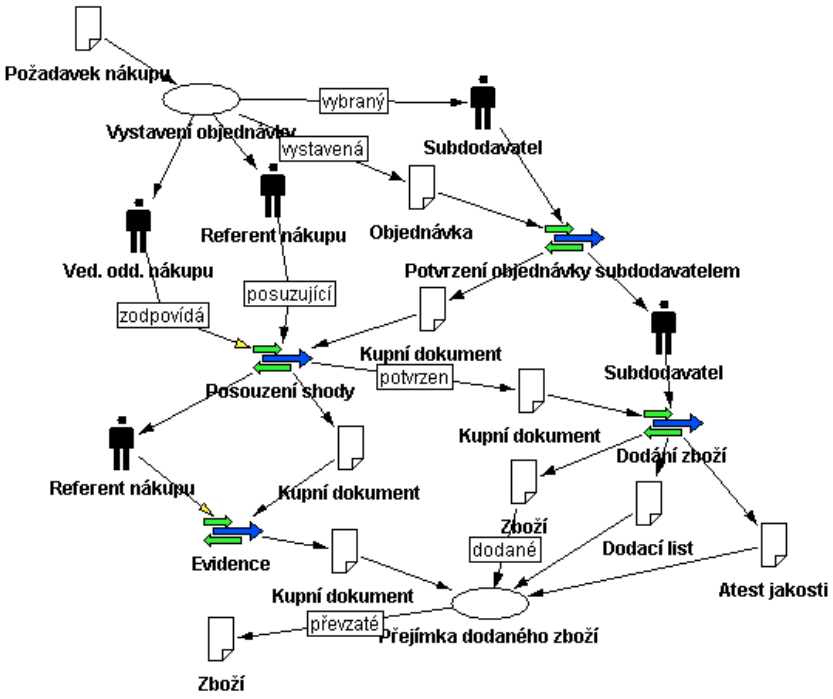
Pomocí této úrovně modelu lze nalézt statickou strukturu všech objektů, které jsou potřebné pro přijetí modelovaného procesu. Objektový pohled vykresluje odpověď na otázku kdo. Snaží se tedy zachytit všechny aktivní předměty, které jsou zodpovědné za provádění činností, a pasivní objekty, které mohou být chápány jako materiál, produkty či dokumenty se kterými je manipulováno.



Obrázek 10: Diagram modelovaný v objektovém pohledu

5.3. Koordinační pohled

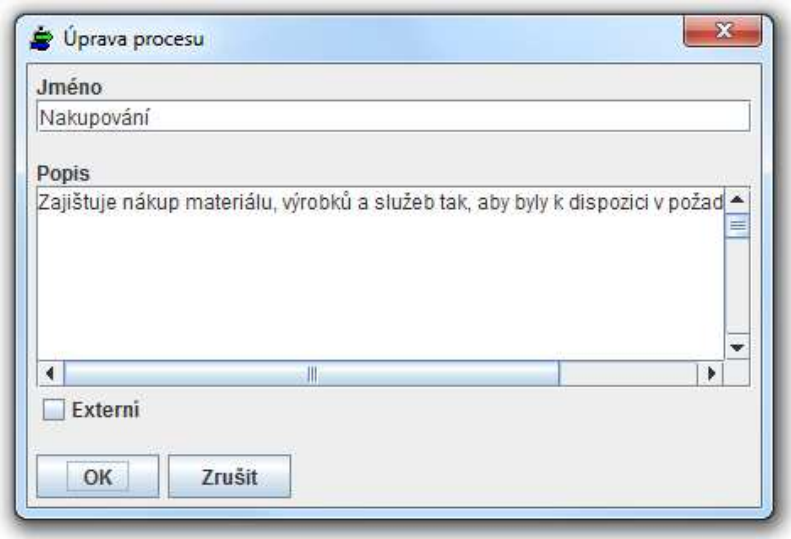
V této úrovni dochází ke sjednocení modelu ve funkčním a objektovém pohledu. Jeho úkolem je ukázat, jak bude modelovaný proces přijat. Koordinační model určuje interakce mezi objekty (aktivní nebo pasivní) a definuje způsob, jak jsou synchronizovány všechny tyto aktivity. Tento pohled je nejdůležitější, protože umožňuje definovat pořadí provádění všech činností, včetně podmínek pro jejich potenciální souběžnost. To znamená, že správné pořadí je definováno stejně jako sdílení použitých zdrojů. Každá činnost má čas a náklady spojené s poskytováním nezbytných informací pro kalkulační analýzu založenou na činnostech.



Obrázek 11: Diagram modelovaný v koordinačním pohledu

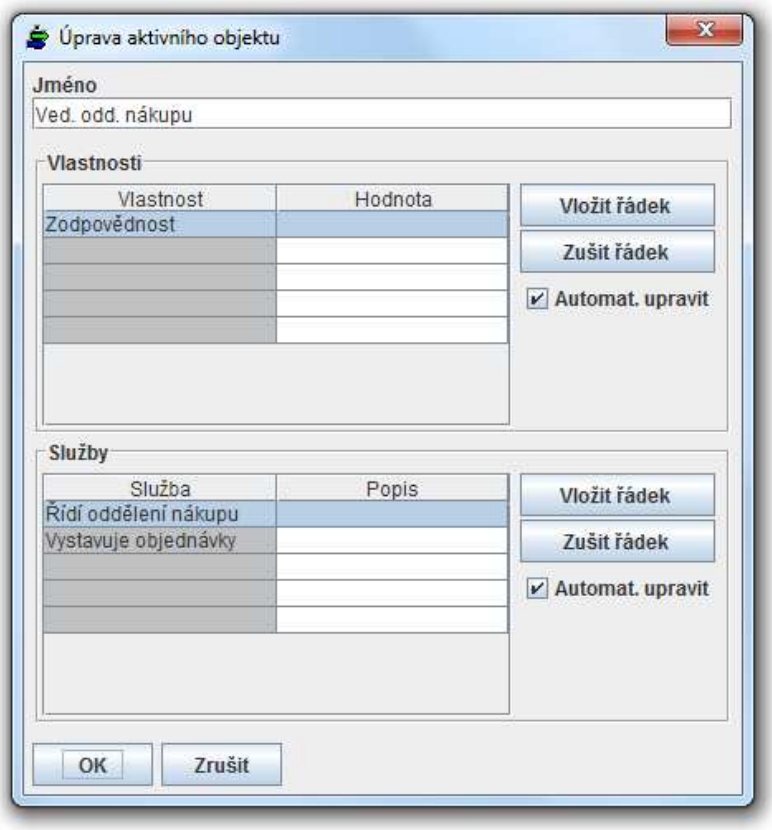
BP Studio obsahuje několik grafických prvků, které je možné využít pro tvorbu modelu podnikového procesu. Základními stavebními prvky jsou: proces, aktivní objekt, pasivní objekt a vazby. Proces představuje jednotlivé činnosti, ze kterých se podnikový proces skládá. Například proces `Nakupování`, který je uveden na obrázku 9, se může skládat z několika úkonů. Může jimi být `Vystavení objednávky` a `Přejímka dodaného zboží`. Tyto jednotlivé činnosti lze poté modelovat v BP Studiu pomocí prvku proces. Dalším prvkem jsou aktivní objekty, které jsou zodpovědné za provádění činností. Příkladem může být `Zákazník` či `Manager`. Pasivní objekty mohou být chápany jako materiál, produkty či dokumenty, se kterými je manipulováno (př. `Objednávka`, `Zboží`). Pomocí vazeb lze vykreslit souvislosti mezi jednotlivými prvky. V BP Studiu existuje několik typů těchto vazeb (Collaboration, Containment, Customer, Owner, Product, Association, Composition, Generalization a Role).

Na obrázku 12 je detail procesu `Nakupování`. U procesu je možné definovat název a popis procesu, dále také volbu, zda je či není proces externí. Externí proces je ten, jehož obsah a jednotlivé kroky zpracování jsou skryty.



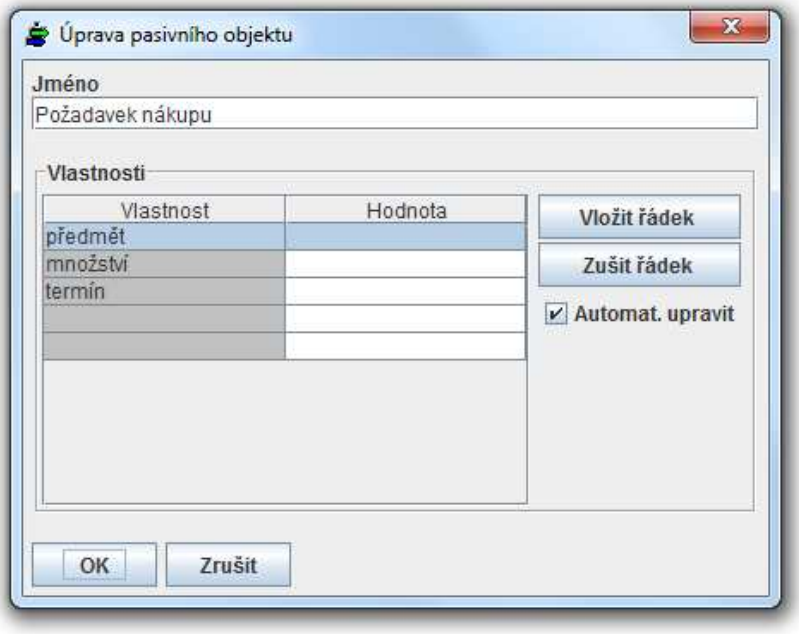
Obrázek 12: Detail procesu "Nakupování"

Aktivní objekt `Vedoucí oddělení nákupu` lze specifikovat uvedením názvu a dále také vlastnostmi a službami. Vlastnost i služba má svůj název a hodnotu. Vlastnosti definují, jaký objekt je a služby udávají aktivity, kterými objekty disponují. V tomto případě lze říci, že aktivní objekt `Vedoucí oddělení nákupu` má zodpovědnost. Tyto vlastnosti nemají uvedeny žádné hodnoty. Dále objekt `Vedoucí oddělení nákupu` řídí oddělení nákupu a vystavuje objednávky.



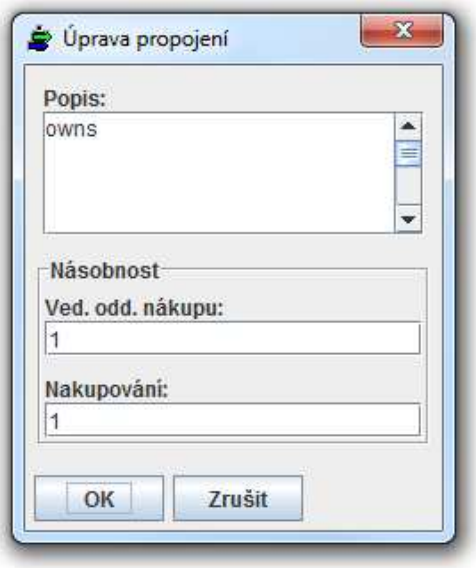
Obrázek 13: Detail aktivního objektu "Vedoucí oddělení nákupu"

Pasivní objekty je možné definovat také pomocí názvu a vlastností. Služby však u tohoto typu prvku nelze uvádět, jelikož se jedná pouze pasivní prvek - dokument, materiál. Na následujícím obrázku je možné vidět pasivní objekt `Požadavek nákupu`. Tento požadavek má vlastnosti jako předmět, množství a termín.



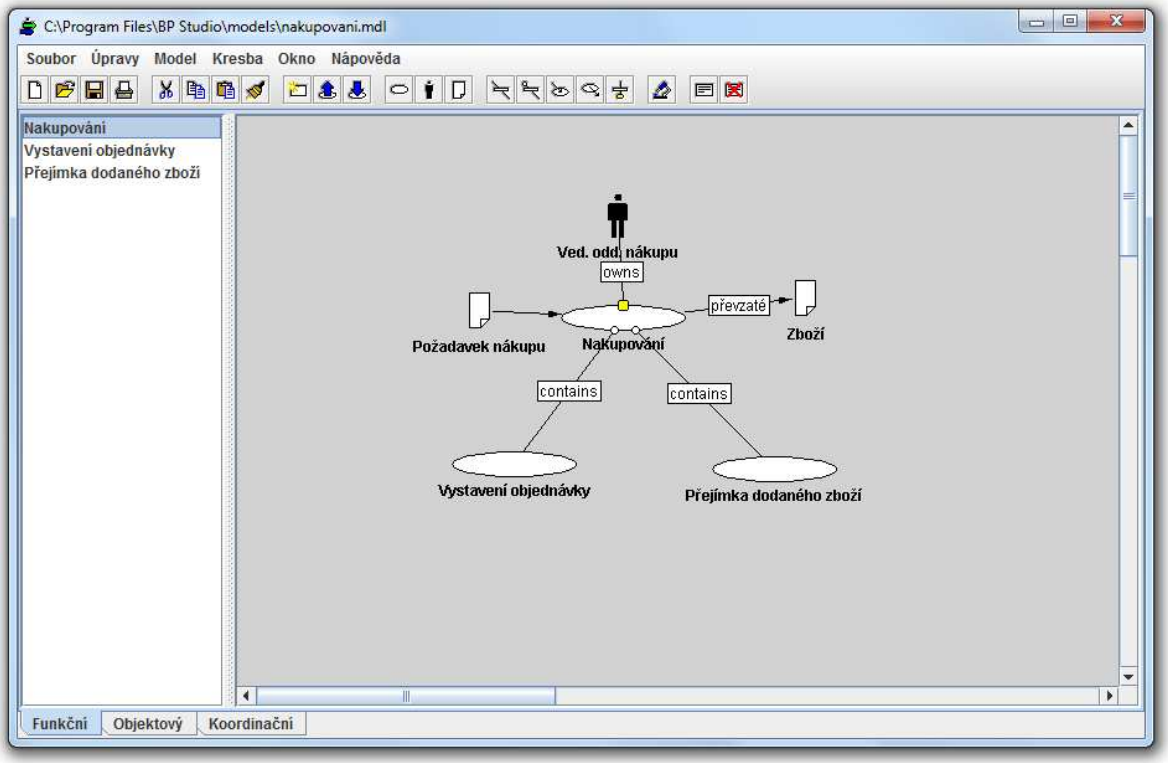
Obrázek 14: Detail pasivního objektu "Požadavek nákupu"

Poslední prvek vazba lze specifikovat vyplněním popisu a také násobností. Násobnost je uvedena vždy od zdrojového prvku ke koncovému. V tomto případě je možné o vazbě říci, že znamená vazbu vlastnictví mezi aktivním objektem `Vedoucím oddělení nákupu` a procesem `Nakupování` a jeho násobnost je 1:1.



Obrázek 15: Detail vazby

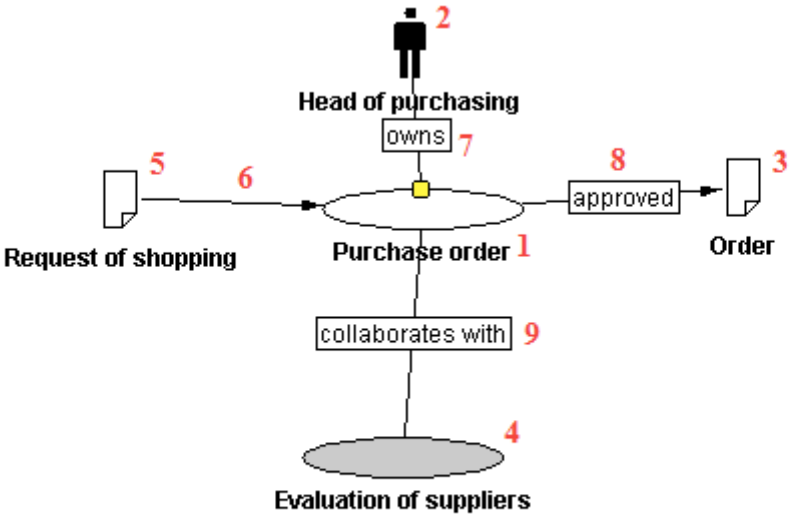
Na následujícím obrázku je zobrazeno grafické uživatelské rozhraní celého modelovacího nástroje BP Studio.



Obrázek 16: Grafické uživatelské rozhraní nástroje BP Studio

5.4. Příklad

Následující příklad je namodelován pomocí BP Studia. Jedná se o součást již dříve uváděného diagramu, avšak z pohledu hlubší úrovně. Představuje detailnější popis procesu `Vystavení objednávky` (`Purchase order`). Tento diagram používá prvky: proces, aktivní objekt, pasivní objekt a vazby (customer, product, owner, collaboration). Příklad představuje proces vystavení objednávky, který je součástí procesu nákupu. Do tohoto procesu vstupuje dokument, který definuje, jak má objednávka vypadat. Dále tento proces spolupracuje s hodnocení dodavatelů, které je vykonáváno externě. Vystavení objednávky řídí vedoucí oddělení nákupu, který současně také tyto objednávky vystavuje. Výsledkem celého procesu je schválená objednávka, která nese své atributy (typ zboží, množství, dodavatel, cena a podobně).



1. **Name: Purchase order**
   Description: Process of purchase order
   Externality: false

2. **Name: Head of purchasing**
   Attribute: Responsibility
   Service: Manage department of purchasing, Create order

3. **Name: Order**
   Attribute: Type of goods, Quantity, Supplier, Cost

4. **Name: Evaluation of suppliers**
   Description: External process
   Externality: true

5. **Name: Request of shopping**
   Attribute: Subject, Quantity

6. **Description: none**

7. **Description: owns**
   Multiplicity from: 1
   Multiplicity to: 1

8. **Description: approved**

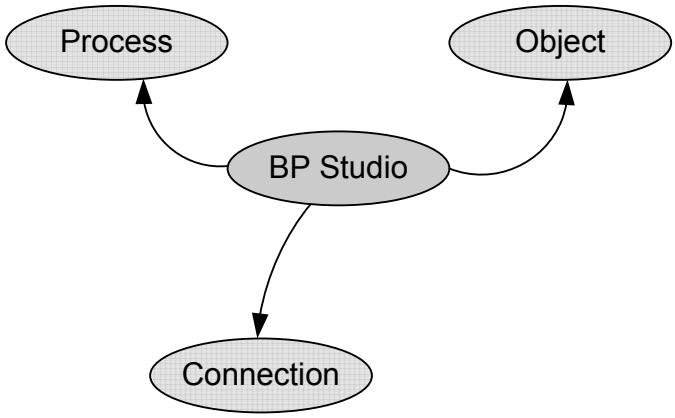
9. **Description: collaborates with**
   Multiplicity from: 1
   Multiplicity to: 1

Obrázek 17: Příklad business procesu "Purchase order"

6. Mapování prvků BP Studia do ontologie

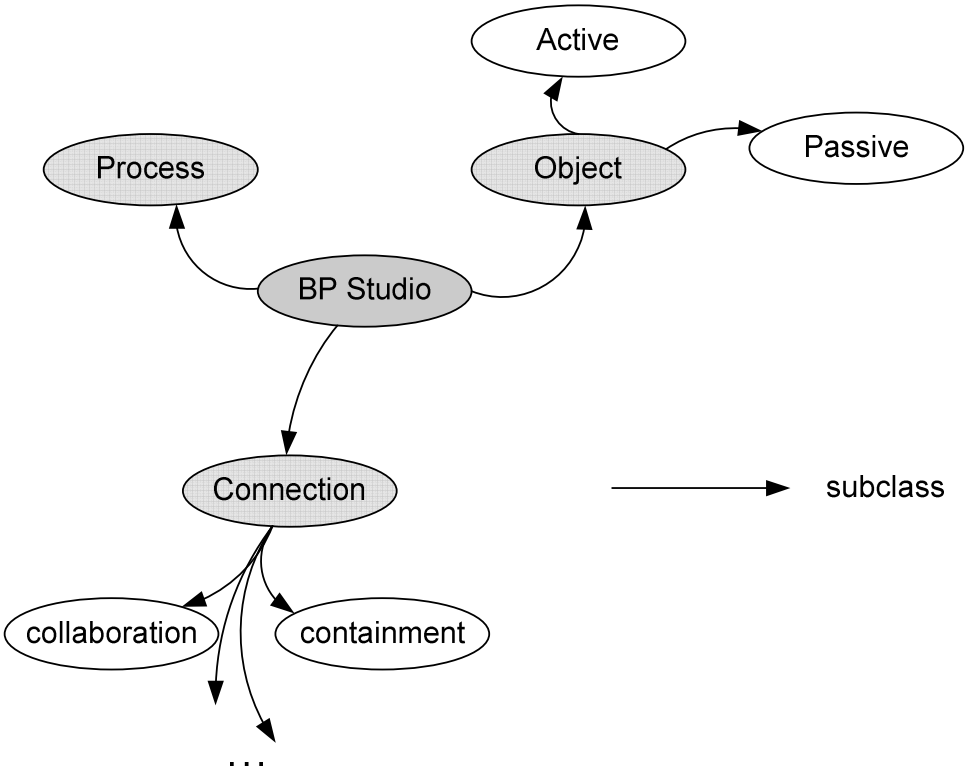
6.1. Třídy

Celá struktura prvků BP Studia lze reprezentovat pomocí ontologie. Tato ontologie obsahuje tři základní třídy, které představují základní grafické prvky BP Studia. Jsou jimi `Process`, `Object` a `Connection`. Tyto třídy jsou disjunktní, jelikož vždy může být prvek pouze jedné třídy. Nemůže se tedy stát, že by byl namodelován prvek, který by byl procesem a zároveň vazbou. Tato struktura je znázorněna na obrázku.



Obrázek 18: Základní ontologické třídy

BP Studio má však dva druhy objektů - aktivní a pasivní. Proto je třída `Object` rozšířena o podtřídy `Active` a `Passive`. Obě tyto podtřídy jsou opět disjunktní, jelikož se nemůže stát, že by jeden objekt byl aktivní a zároveň i pasivní. Stejně tak má BP Studio řadu vazeb. Třída `Connection` je tedy rozšířena o jednotlivé typy vazeb (`Collaboration`, `Containment`, `Customer`, `Owner`, `Product`, `Association`, `Composition`, `Generalization`, `Role`) a opět je u všech těchto typů nastavena disjunktnost. Následující obrázek zobrazuje tyto změny, avšak u typů vazeb jsou pro zjednodušení uvedeny pouze jen některé z nich.



Obrázek 19: Rozšíření ontologických tříd

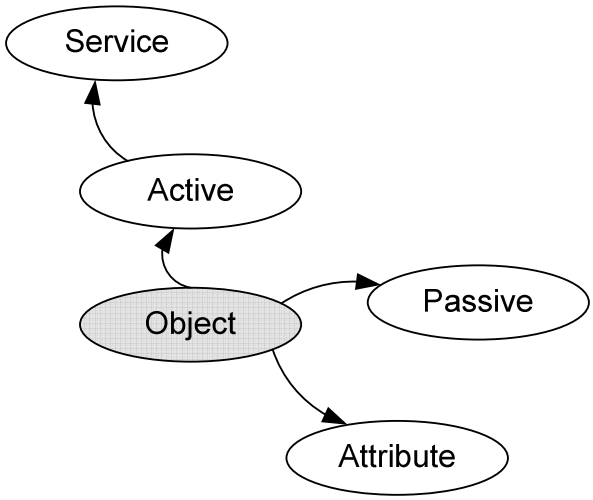
V této fázi má ontologie základní strukturu. V BP Studiu však kromě vytváření jednotlivých elementů je možné také tyto prvky různě specifikovat. Každý prvek tohoto modelovacího nástroje má své atributy.

U aktivního i pasivního objektu lze nadefinovat vlastnosti (attribute). U aktivního objektu lze dále nastavit i služby (service). Vlastnosti definují, jaký objekt je a služby jsou činnosti, které mají objekty na starosti nebo kterými disponují. Příkladem vlastnosti pasivního objektu `Objednávka` může být `Číslo objednávky`, `Množství` nebo například `Typ zboží`. Služby aktivního objektu `Manager` mohou být například `Řízení projektu`, `Zadávání projektů` a podobně.

Jelikož vlastnosti i služby mohou obsahovat název i hodnotu či popis, bylo jednodušší vytvořit samostatné třídy pro tyto atributy, aby mohly vznikat jedinci představující jednotlivé vlastnosti a služby. Vznikly tak tedy třídy `Attribute` a `Service`. Jelikož nastavení vlastností mají aktivní i pasivní objekty totožné, v ontologii je třída `Attribute` podtřídou `Object`. To znamená, že třída `Active` i `Passive` může obsahovat vazby na třídu `Attribute`. Jinými slovy, aktivní i pasivní objekt může mít nadefinovány své vlastnosti.

V tomto případě není nastavena disjunktnost, jelikož modelovaný prvek může být aktivním či pasivním objektem a zároveň může mít také své vlastnosti. Aktivní objekt, jak už bylo výše zmíněno, obsahuje i definici služeb. V tomto případě je však třída `Service` zařazena v ontologii pouze jako

podtřída třídy `Active`, jelikož u pasivního prvku se služby nenastavují. Pro lepší přehlednost je vše znázorněno na obrázku 20.



Obrázek 20: Třídy Attribute a Service

6.2. Datové vlastnosti

V BP Studiu existuje řada dalších vlastností jednotlivých prvků, které dosud není možné se současnou verzí ontologie definovat.

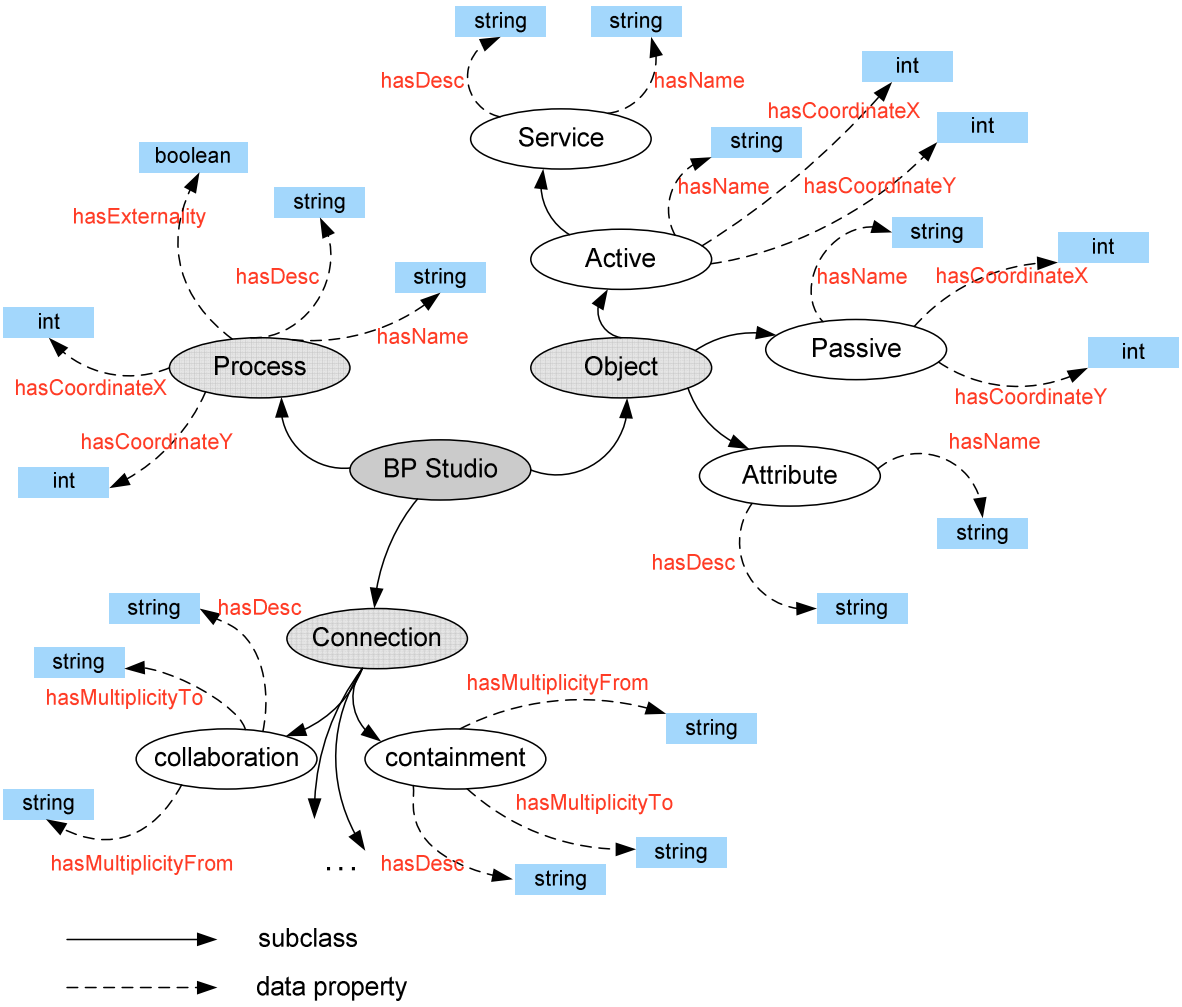
U prvku proces je to název, popis a volba, zda je či není proces externí. Tyto vlastnosti se v ontologii reprezentují pomocí tzv. datových vlastností. Pro definování názvu procesu je vytvořena datová vlastnost `hasName`, pro popis je to `hasDesc` a pro volbu, zda je externí, `hasExternality`.

Aktivní objekt má název a poté vlastnosti a služby. Pro název opět slouží datová vlastnost `hasName`. Vlastnosti a služby jsou již definovány třídami, ale každá má název a hodnotu. Pro název vlastnosti i služby se tedy využije opět datová vlastnost `hasName` a pro určení hodnoty vlastnosti a služby `hasDesc`. Pasivní objekt je obdobný jako aktivní, akorát s tím rozdílem, že neobsahuje služby.

Vazba může obsahovat popis a násobnost. Popis je v ontologii reprezentován vlastností `hasDesc`. U násobnosti je nutné uvést hodnotu zdrojového i koncového prvku. Příkladem může být násobnost 1:1, 1:N nebo M:N. Proto v ontologii existují dvě datové vlastnosti, a to `hasMultiplicityFrom` a `hasMultiplicityTo`. Mnohonásobnost se určuje pouze u některých typů vazeb.

Jelikož se tato práce zabývá i zpětným nahráváním ontologie do BP Studia, je zapotřebí definovat i grafické pozice jednotlivých prvků, aby mohlo být vše řádně vykresleno. K tomu poslouží datové vlastnosti `hasCoordinateX` a `hasCoordinateY`, které představují horizontální a vertikální souřadnice jednotlivých grafických prvků.

Veškeré definované datové vlastnosti jsou znázorněny na následujícím obrázku 21. Datové vlastnosti vždy nabývají nějaké hodnoty, a proto je vždy u každé z nich uveden také datový typ.



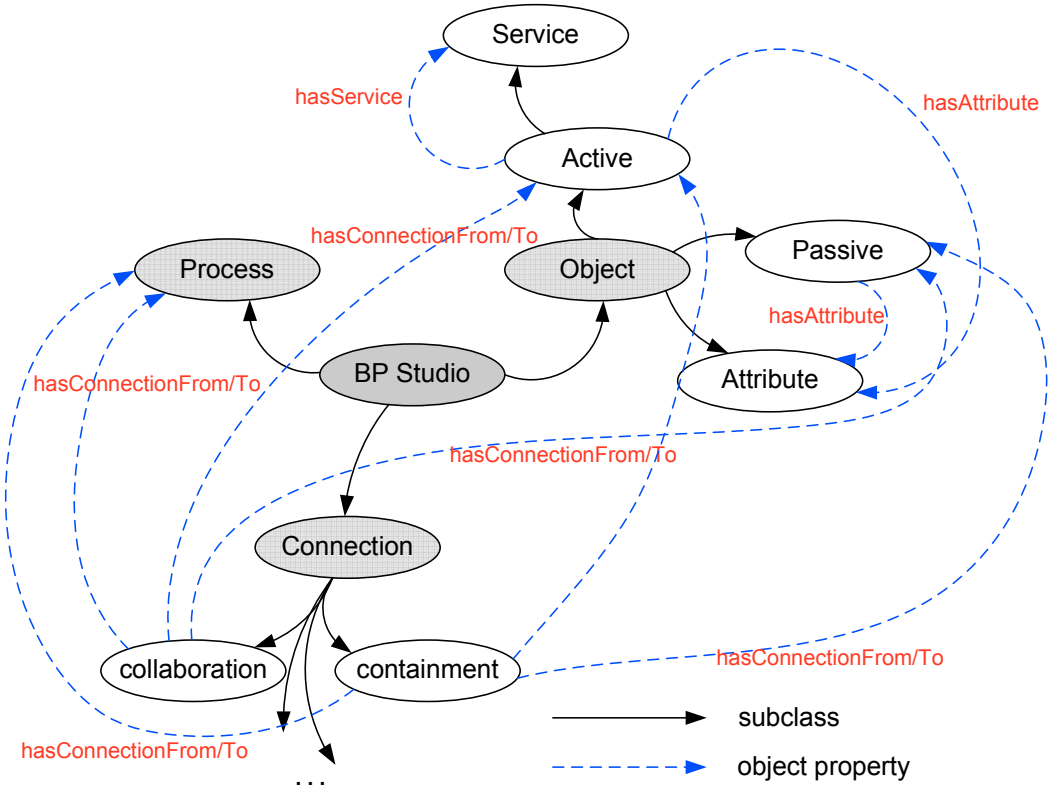
Obrázek 21: Datové vlastnosti ontologie

6.3. Objektové vlastnosti

Tato definice ontologie však ještě není zcela kompletní. V současné fázi ontologie je možné uložit jednotlivé prvky, společně s jejich vlastnostmi. Není však dosud nikde definováno spojení mezi těmito prvky. Tyto vztahy jsou reprezentovány pomocí objektových vlastností. Díky těmto vlastnostem dochází k propojení mezi třídami, resp. mezi jedinci tříd.

Samozřejmě že spojení mezi prvky reprezentuje třída `Connection`. Její jedinci obsahují informace o popisu a násobnosti vazby, ale je také nutné definovat pomocí objektové vlastnosti, které prvky vlastně spojují. K tomu poslouží objektové vlastnosti `hasConnectionFrom` a `hasConnectionTo`. Jejich hodnotami poté budou konkrétní jedinci.

Dalšími objektovými vlastnostmi, které je nutné vytvořit, jsou vazby mezi jedinci třídy `Active-Attribute`, `Active-Service` a `Passive-Attribute`. Dosud by totiž došlo pouze k vytvoření jednotlivých jedinců tříd, nikde by však nebyla definována souvislost mezi jedinci. Tyto objektové vlastnosti se nazývají `hasAttribute` a `hasService`.



Obrázek 22: Objektové vlastnosti ontologie

6.4. Definiční obor a obor hodnot

Pro přesnější definování ontologie slouží specifikace omezení, které zabrání nekonzistenci ontologie a vytvoří ji jednoznačnou. K tomu slouží globální omezení. To lze nastavit pomocí definování definičních oborů a oborů hodnot. Tato dvě kritéria se nastavují u jednotlivých vlastností a říkají, kteří jedinci (třídy) mohou konkrétní vlastnosti využívat. Tak například je zcela zřejmé, že objektovou vlastnost `hasAttribute` nebude využívat třída `Process`, jelikož procesu nelze nastavit žádné vlastnosti. Při strojovém zpracování však toto zřejmé není a je tedy nutné ohraničit použití dané objektové vlastnosti. Toho lze docílit pomocí nastavení definičního oboru (domain), který představuje množinu tříd, které mohou vlastnost využít. Je také vhodné nastavit obor hodnot (range), který definuje množinu hodnot, kterých může vlastnost nabývat. U datových vlastností lze samozřejmě také nastavit definiční obor a obor hodnot. Obor hodnot však v tomto případě nebude množina tříd, ale pouze datový typ, kterého může daná vlastnost nabývat. Veškeré vlastnosti a jejich definiční obory a obory hodnot jsou uvedeny v tabulce 1.

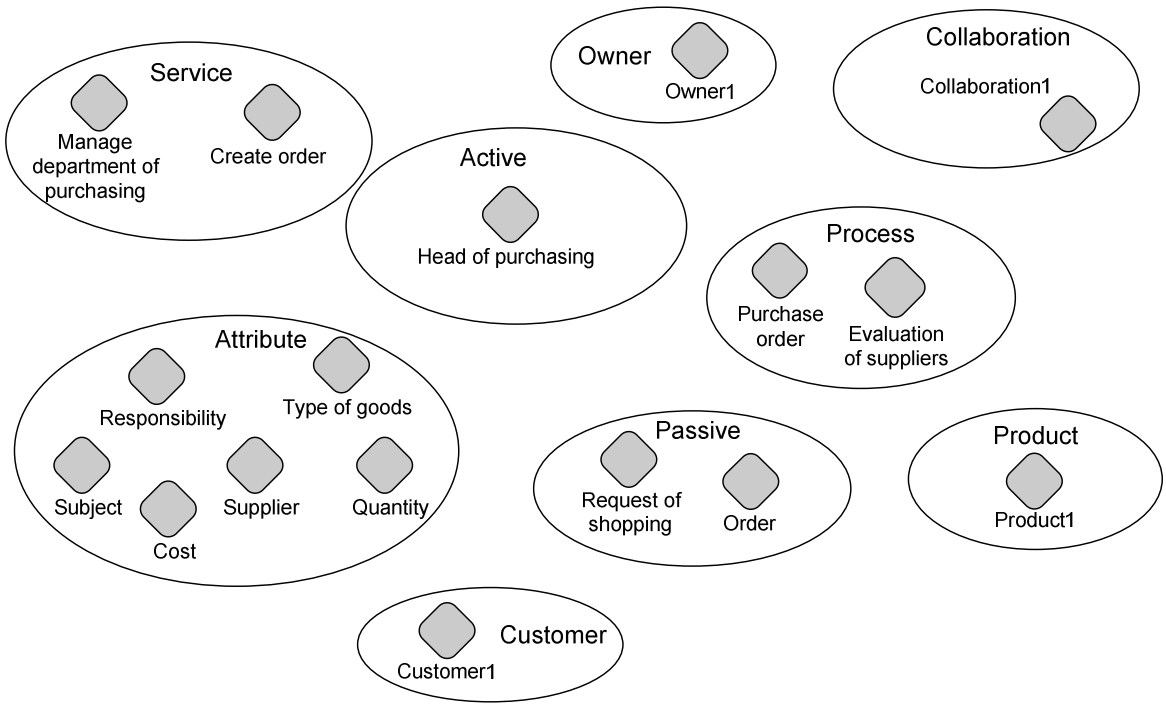
Vlastnosti	**Definiční obor $D_f()$**	**Obor hodnot $H_f()$**
hasAttribute	Active, Passive	Attribute
hasService	Active	Service
hasConnectionFrom	všechny typy Connection	Process, Active, Passive
hasConnectionTo	všechny typy Connection	Process, Active, Passive
hasCoordinateX	Process, Active, Passive	integer
hasCoordinateY	Process, Active, Passive	integer
hasDesc	Process, Attribute, Service, všechny typy Connection	string
hasExternality	Process	boolean
hasName	Process, Active, Passive, Attribute, Service	string
hasMultiplicityFrom	všechny typy Connection	string
hasMultiplicityTo	všechny typy Connection	string

Tabulka 1: Přehled definičních oborů a oborů hodnot

6.5. Použití příkladu

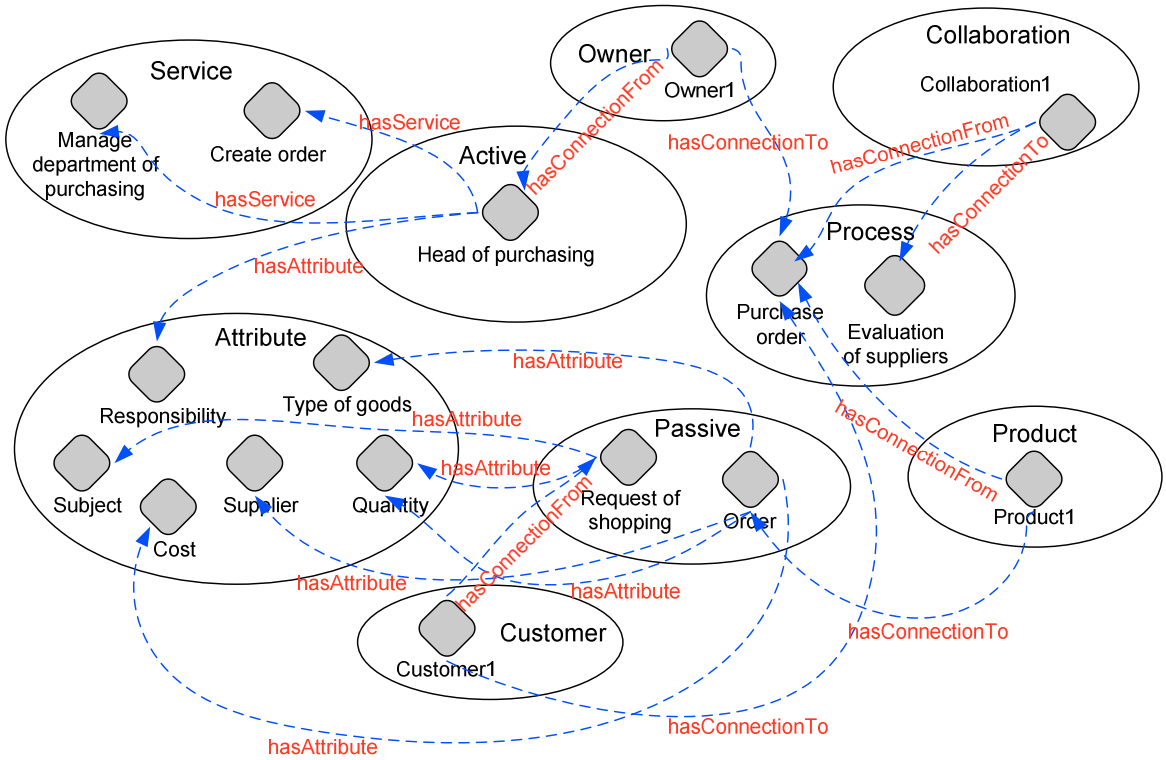
Pomocí již dříve uvedeného příkladu `Vystavení objednávky` (`Purchase order`) je možné názorně demonstrovat převedení prvků BP Studia do ontologie.

V prvním kroku dochází k vytváření jedinců jednotlivých tříd. V tomto případě se jedná o jedince třídy `Process` (`Purchase order` a `Evaluation of suppliers`), jedince třídy `Active` (`Head of purchasing`) a jedince třídy `Passive` (`Request of shopping` a `Order`). Dále je potřeba vytvořit také jedince pro třídu `Attribute` (`Responsibility`, `Type of goods`, `Quantity`, `Supplier`, `Subject` a `Cost`) a pro třídu `Service` (`Manage department of purchasing` a `Create order`). Dále je také potřeba definovat jedince třídy `Collaboration`, `Owner` a `Customer` a `Product` pro jednotlivé vazby. Na obrázku 23 je popsaná konstrukce znázorněna.



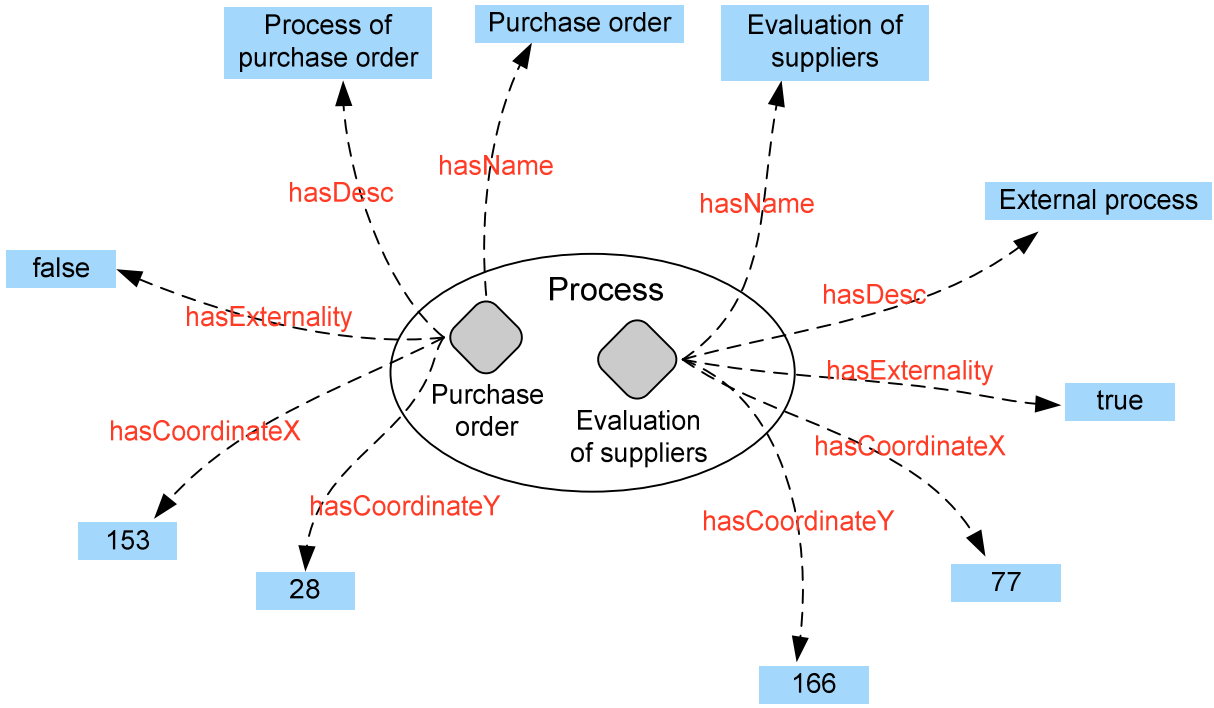
Obrázek 23: Příklad - vytvoření jedinců a přiřazení do tříd

Dále se definují datové a objektové vlastnosti jednotlivých prvků. Datové vlastnosti v této ontologii představují jednotlivé atributy prvků. Příkladem může být proces `Purchase order`, jehož název ("Purchase order"), popis ("Process of purchase order") a volba externality (false) jsou definovány právě datovými vlastnostmi. Objektové vlastnosti v této ontologii slouží k propojení jednotlivých jedinců a tudíž určují celou strukturu ontologie. Na obrázku 24 lze vidět schéma příkladu uloženého pomocí ontologie. Modré šipky představují objektové vlastnosti a červené popisky jejich názvy.



Obrázek 24: Příklad - objektové vlastnosti

Znázornění datových vlastností je zredukováno pouze na jedince třídy `Process`, jelikož zobrazení vlastností všech prvků by bylo velice rozsáhlé a nepřehledné.



Obrázek 25: Detail třídy Process - datové vlastnosti

Následující kód je ukázkou definice procesu `Purchase order`. Nejprve je tedy vytvořen jedinec třídy `Process` a poté jsou definovány datové vlastnosti prvku.

```
1  <ClassAssertion>
2     <Class IRI="#Process"/>
3     <NamedIndividual IRI="C:/Users/mou038/Process1328184291769"/>
4  </ClassAssertion>
5  <DataPropertyAssertion>
6     <DataProperty IRI="#hasDesc"/>
7     <NamedIndividual IRI="C:/Users/mou038/Process1328184291769"/>
8     <Literal datatypeIRI="http://www.w3.org/2001/XMLSchema#string">Process
      of purchase order</Literal>
9  </DataPropertyAssertion>
10 <DataPropertyAssertion>
11    <DataProperty IRI="#hasExternality"/>
12    <NamedIndividual IRI="C:/Users/mou038/Process1328184291769"/>
13    <Literal
      datatypeIRI="http://www.w3.org/2001/XMLSchema#boolean">false</Literal>
14 </DataPropertyAssertion>
15 <DataPropertyAssertion>
16    <DataProperty IRI="#hasName"/>
17    <NamedIndividual IRI="C:/Users/mou038/Process1328184291769"/>
18    <Literal datatypeIRI="http://www.w3.org/2001/XMLSchema#string">Purchase
      order</Literal>
19 </DataPropertyAssertion>
```

Složitější definici je možné vidět u prvku `Request of shopping`, který kromě názvu "Request of shopping" obsahuje i vlastnosti "Subject" a "Quantity". V tomto případě dochází k vytvoření jedinců třídy `Passive` (`Request of shopping`) i `Attribute` (`Subject` a `Quantity`) na řádcích 1-12 a jejich definici datových vlastností na řádcích 25-39. Dále je také nutné využít i objektových vlastností `hasAttribute` a propojit tyto jedince (14-23).

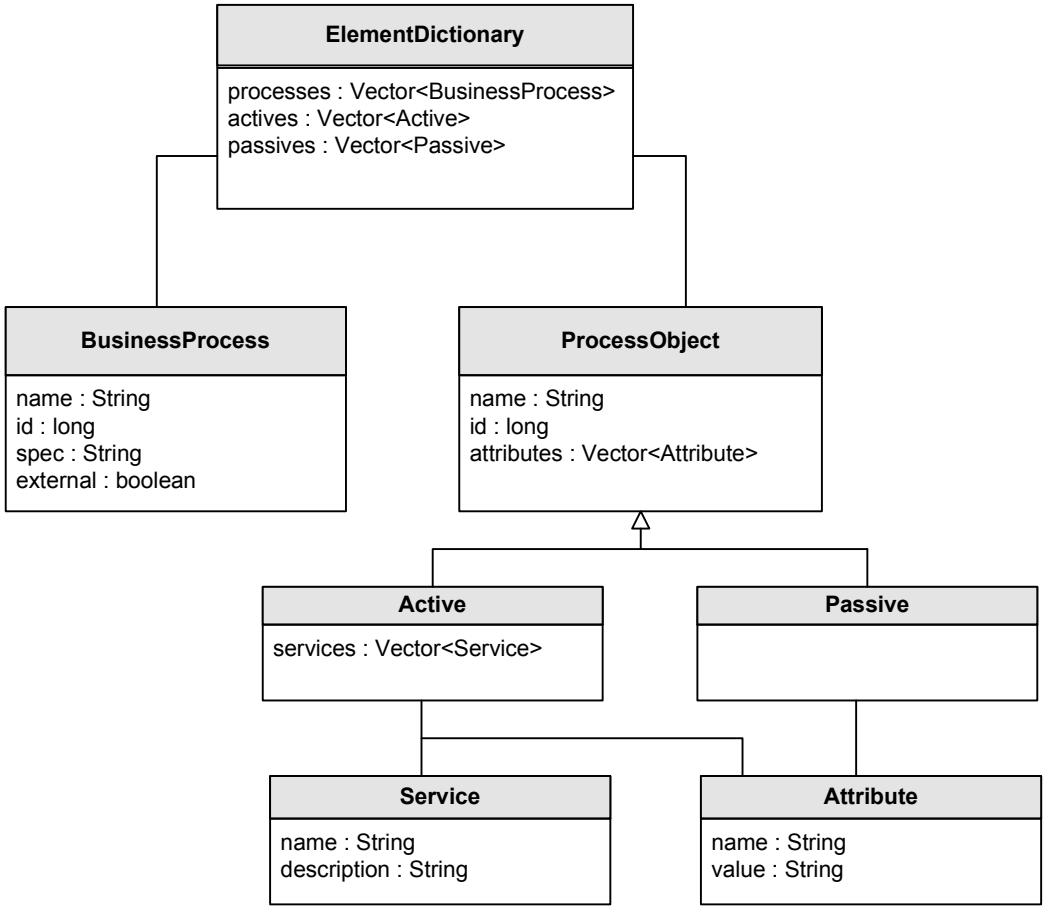
```
1   <ClassAssertion>
2      <Class IRI="#Passive"/>
3      <NamedIndividual IRI="C:/Users/mou038/Passive1218177291378"/>
4   </ClassAssertion>
5   <ClassAssertion>
6      <Class IRI="#Attribute"/>
7      <NamedIndividual IRI="C:/Users/mou038/Attribute-1-1218177291378"/>
8   </ClassAssertion>
9   <ClassAssertion>
10     <Class IRI="#Attribute"/>
11     <NamedIndividual IRI="C:/Users/mou038/Attribute-2-1218177291378"/>
12  </ClassAssertion>
13
14  <ObjectPropertyAssertion>
15    <ObjectProperty IRI="#hasAttribute"/>
16    <NamedIndividual IRI="C:/Users/Tereza/Attribute-1-1218177291378"/>
17    <NamedIndividual IRI="C:/Users/Tereza/Passive1218177291378"/>
18  </ObjectPropertyAssertion>
19  <ObjectPropertyAssertion>
20    <ObjectProperty IRI="#hasAttribute"/>
21    <NamedIndividual IRI="C:/Users/Tereza/Attribute-2-1218177291378"/>
22    <NamedIndividual IRI="C:/Users/Tereza/Passive1218177291378"/>
23  </ObjectPropertyAssertion>
24
```

```
<DataPropertyAssertion>
  <DataProperty IRI="#hasName"/>
  <NamedIndividual IRI="C:/Users/Tereza/Attribute-1-1218177291378"/>
  <Literal datatypeIRI="http://www.w3.org/2001/XMLSchema#string">Subject</Literal>
</DataPropertyAssertion>
<DataPropertyAssertion>
  <DataProperty IRI="#hasName"/>
  <NamedIndividual IRI="C:/Users/Tereza/Attribute-2-1218177291378"/>
  <Literal datatypeIRI="http://www.w3.org/2001/XMLSchema#string">Quantity</Literal>
</DataPropertyAssertion>
<DataPropertyAssertion>
  <DataProperty IRI="#hasName"/>
  <NamedIndividual IRI="C:/Users/Tereza/Passive1218177291378"/>
  <Literal datatypeIRI="http://www.w3.org/2001/XMLSchema#string">Request of shopping</Literal>
</DataPropertyAssertion>
```

7. Implementace

7.1. Struktura ukládání dat

Data nástroje BP Studio jsou uložena pomocí třídy `ElementDictionary`, která představuje tzv. slovník prvků. Jsou zde uložena data, týkající se procesů, aktivních a pasivních prvků. Výjimkou jsou vazby, které jsou uloženy pouze na úrovni grafických prvků. Třída `ElementDictionary` obsahuje seznamy instancí jednotlivých typů prvků. Každý druh prvku (proces, aktivní objekt, pasivní objekt) je uložen ve své datové struktuře typu `Vector`. Prvky mají definovány své vlastní třídy. Tyto třídy jsou uloženy v balíčku `bpm.method`. Základní třídou je `ProcessElement`, která je dále rozvedena do jednotlivých typů prvků: `BusinessProcess` (proces), `Active` (aktivní objekt), `Passive` (pasivní objekt). Jednotlivé třídy nesou nezbytné informace o daném prvku, jako je identifikační číslo, název, popis a další atributy. Tato struktura je znázorněna pomocí obrázku 26.

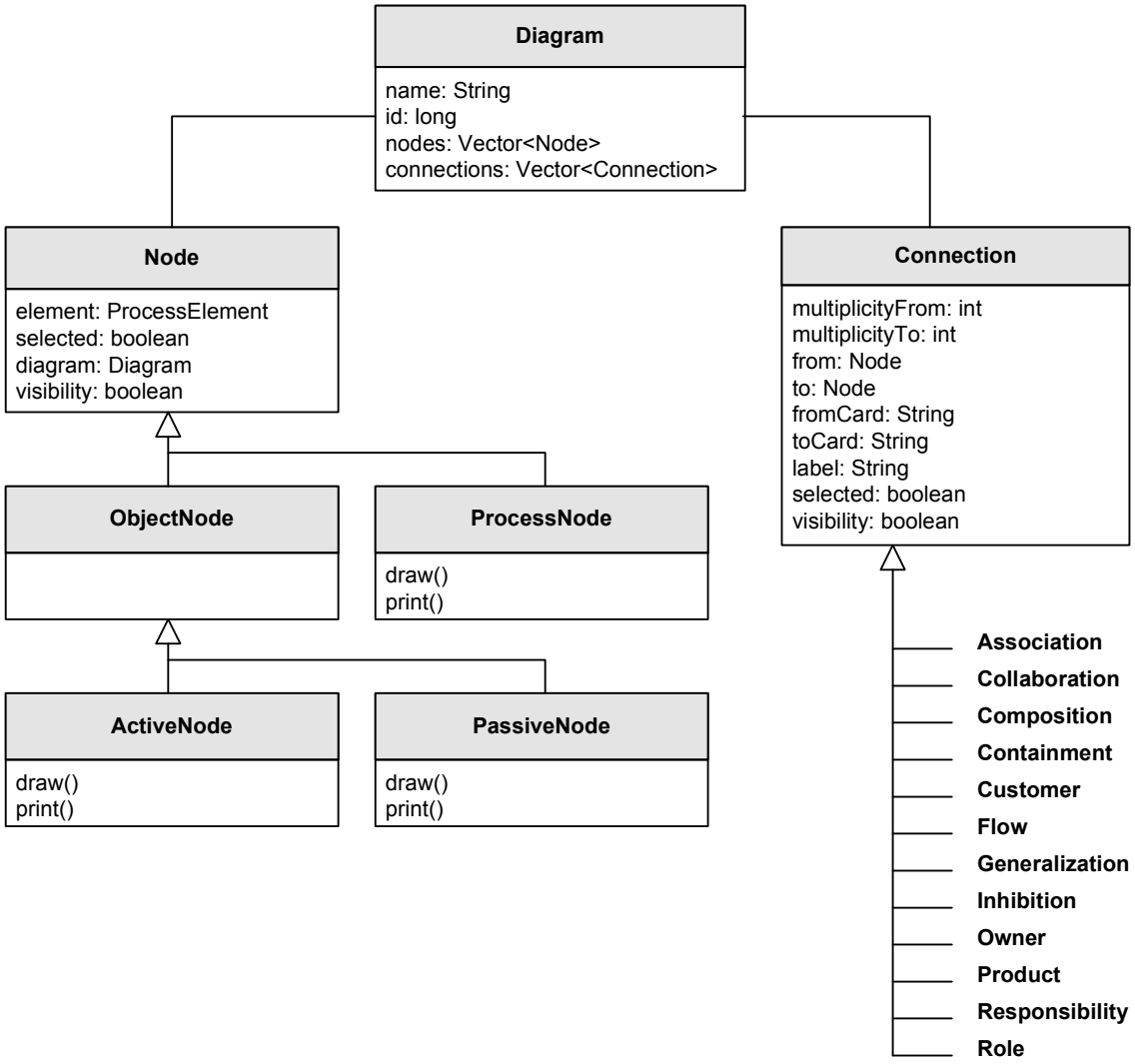


Obrázek 26: Struktura pro ukládání do slovníku prvků

Do struktury ElementDictionary se prvky zapisují hned při jejich vytvoření a jsou k dispozici po celou dobu, dokud nejsou zase z této struktury smazány. To znamená, že smazání prvku z hlavního okna nemá žádný vliv na strukturu ElementDictionary. Prvky zobrazené v hlavním okně se totiž ukládají do samostatné struktury určené pouze pro grafické zobrazení a obsahují pouze odkaz na prvky z ElementDictionary.

Struktura pro grafické zobrazení prvků se vytváří ve chvíli kdy je do hlavního okna vložen jakýkoli prvek a doplňuje se či mění v závislosti na dalším vkládání. Jednotlivé třídy definující grafické prvky jsou uloženy v balíčku bpm.drawing. Celý diagram je poté uložen pomocí datové struktury Vector<Diagram> ve třídě ModelDictionary. Tato třída obsahuje tři diagramy, a to funkční, objektový a koordinační, které korespondují s jednotlivými pohledy BP Studia. Jinými slovy lze říci, že každý pohled má svůj diagram.

Třída Diagram představuje jednotlivé diagramy namodelované v hlavním okně. Diagramy se od sebe rozlišují pomocí jedinečného identifikátoru, jelikož může být v jednom pohledu i více diagramů. Každý diagram dále nese své jméno a také seznam jednotlivých prvků a spojení, která jsou v diagramu namodelována. Prvky jsou uloženy ve struktuře Vector<Node> a vazby pomocí Vector<Connection>. Třída Node definuje jednotlivé prvky po grafické stránce. Obsahuje odkaz na prvek pomocí instance třídy ProcessElement a informaci o tom, ve kterém diagramu je prvek uložen. Třída Node se dále rozděluje na Object Node a ProcessNode. ObjectNode se dále dělí na ActiveNode a PassiveNode. Všechny tyto třídy vytváří grafickou podobu prvků a následně je zobrazují. Vazby jsou ve třídě Diagram uloženy pomocí třídy Connection. Třída Connection definuje jednotlivé parametry vazby, jako je mnohonásobnost, popis, kardinalita a především odkaz na prvky, které tato vazba spojuje. Třída Connection je dále rozdělena na jednotlivé typy vazeb. Tyto třídy nastavují vazbě svůj typ a nakonec její finální podobu v závislosti na daném typu. Struktura tříd pro grafické zobrazení je znázorněna na obrázku 27.



Obrázek 27: Grafická struktura ukládání dat

7.2. Získání dat

Pro získání jednotlivých instancí prvků slouží metoda `getDiagram()` třídy `ExtractModel` (balíček `bpm.builder`). V této metodě se nejprve načte do datové struktury `Vector<Diagram>` funkční diagram pomocí instance třídy `Method`, která má příslušné metody pro získání dat.

Do pole typu `Vector<Node>` se načtou jednotlivé prvky diagramu. Toto pole se poté prochází a rozřazuje prvky do jednotlivých typů, jako jsou proces, aktivní a pasivní objekt. Do struktury diagramu se však ukládá pouze jednoznačný identifikátor elementu, jelikož se jedná pouze o datovou strukturu pro grafické zobrazení. Veškeré další informace o jednotlivých prvcích jsou uloženy ve struktuře `ElementDictionary` a proto je ještě nutné tato data získat pomocí metod `setProcessData(long id, Point point)`, `setActiveData(long id, Point point)` a `setPassiveData(long id, Point point)`. Příkladem může být metoda

`setProcessData(long id, Point point)`, která pomocí vstupního parametru - jednoznačného identifikátoru dokáže nalézt potřebné doplňující informace. Součástí této metody je také dodatečné nastavení souřadnic prvku, které později slouží ke zpětnému vykreslení diagramu. Nastavení souřadnic bylo do tříd prvků přidáno z toho důvodu, aby všechny potřebné informace pro ontologii byly na jednom místě.

```
1   public Vector getDiagram() {
2      Vector<Diagram> diagrams = method.getDiagrams(DiagramType.FUNCTIONAL);
3      for(Diagram diagram : diagrams) {
4         Vector<Node> nodes = diagram.getNodes();
5         for(Node node : nodes) {
6            ProcessElement element = node.getElement();
7            if(element.getType().equals("processes")) {
8               setProcessData(element.getId(), node.getOrigin());
9            } else if(element.getType().equals("active")) {
10               setActiveData(element.getId(), node.getOrigin());
11           } else if(element.getType().equals("passive")) {
12              setPassiveData(element.getId(), node.getOrigin());
13           }
14        }
15     }
16  }
17  public void setProcessData(long id) {
18     Vector<BusinessProcess> vProcesses = method.getProcesElements();
19     for(BusinessProcess process : vProcesses) {
20        if(process.getId() == id) {
21           process.setPoint(getCoordinates("process",process.getId()));
22           vProcess.add(process);
23        }
24     }
25  }
```

Vazby jsou v grafické struktuře ukládány do `Vectoru` třídy `Connection`. Tento seznam spojení lze získat pomocí již načteného funkčního diagramu. Pro zjištění prvků, které vazba spojuje však třída `Connection` vrací pouze instance třídy `Node`. Pro účely této diplomové práce je však nutné získat také jednoznačné identifikátory těchto prvků. Aby byla opět veškerá potřebná data pohromadě, došlo k vytvoření třídy `Binding` v balíčku `bpm.builder`, která tato data sjednocuje a představuje tedy v tomto případě prvek vazby. Instance třídy `Binding`, doplněná již o všechny informace, se následně uloží do datové struktury typu `Vector<Binding>`.

Extrahovaná data jsou ukládána do vektorových polí jednotlivých typů prvků. Tato pole jsou poté uložena do dalšího vektorového pole, aby došlo ke zjednodušení předávání parametrů.

7.3. Export do ontologie

Pro export diagramu do ontologie slouží třída `BuilderOntology` v balíčku `bpm.builder`. Tato práce využívá OWL API knihovnu pro vytváření ontologií. Vektorové pole, které obsahuje jednotlivé seznamy prvků, je v této třídě extrahováno.

Nejprve je však nutné vytvořit OWL managera, který následně nahraje připravenou šablonu ontologie. Tato šablona obsahuje strukturu tříd, datových a objektových vlastností, které jsou popsány v kapitole BP Studio. Dalším krokem je vytvoření lokální ontologie, která bude sloužit jako pracovní verze. Díky OWL manageru lze jednoduše obsluhovat načtenou ontologii. Dále je také nutné vytvořit `DataFactory`, které slouží pro vytváření jednotlivých jedinců. Popsané úkony jsou součástí metody `loadTemplate()`, která se společně s metodou `loadData()`, volá již v konstruktoru. Metoda `loadData()` načítá strukturu tříd a množiny vlastností dané šablony.

Ve třídě `BuilderOntology` se dále vyskytují metody pro nastavení jednotlivých parametrů prvku. Metoda `setType(String type)` nastavuje typ prvku, který může nabývat hodnot (`Process`, `Active`, `Passive`, `Service`, `Attribute`, a jednotlivé typy vazeb). Tento parametr tedy určuje, do které třídy bude jedinec vytvářen. Dále je zde metoda `setId(long id)`, která nastavuje a zároveň také rovnou vytváří jedince daného prvku. Jedinec je do třídy přidán pomocí třídy `OWLClassAssertionAxiom`.

Pojmenování jedince vychází ze spojení typu prvku a jeho identifikačního čísla, aby byla zajištěna jeho jednoznačnost. To znamená, že například prvek typu "Process" s id = 1234 bude mít název Process1234. U jedinců typu `Attribute` a `Service` dochází k poněkud odlišnému pojmenování, jelikož se nejedná o samostatné prvky, ale pouze o součást aktivního nebo pasivního objektu, a tudíž nemají ani svá vlastní id. Jejich název se skládá z typu jedince, pořadového čísla, které je nastaveno pomocí metody `setOrder(int order)` a id aktivního nebo pasivního prvku, ke kterému patří. Příkladem může být název jedince třídy `Service`. Služba je součástí aktivního prvku s id = 5678. V tomto případě bude název jedince vypadat takto: Service-1-5678. V případě, že by aktivní prvek obsahoval více služeb, jejich pořadové číslo bude postupně narůstat.

V následujícím příkladě je uvedena metoda `setId(long id)`. Metoda nejprve nalezne ontologickou třídu, která je definovaná pomocí proměnné type. V případě, že se jedná o prvek typu `Service` nebo `Attribute`, je potřeba do názvu zakomponovat také pořadí (`order`), jak již bylo vysvětleno v předchozím odstavci. Jinak je název tvořen klasickým způsobem. Dále je vytvořena vazba mezi třídou a jedince a poté vše přidáno do ontologie pomocí manageru.

```
1    public void setId(long id) {
2       this.id = id;
3       for(OWLClass cl : classes) {
4          if (cl.toString().contains(type)) {
5             if(type.equals("Service")||type.equals("Attribute")) {
6                individual = dataFactory.getOWLNamedIndividual(":" + type + "-" +
                 order + "-" + id, pm);
7             } else {
8                individual = dataFactory.getOWLNamedIndividual(":" + type + id, pm);
9                temp = individual;
10            }
11            classAssertion = dataFactory.getOWLClassAssertionAxiom(cl, individual);
```

```
12          }
13       }
14       manager.addAxiom(localBPStudio, classAssertion);
15    }
```

Bylo nutné vytvořit samostatnou metodu `setIdConnection()` pro vytváření prvků vazby, jelikož vazby nemají svá identifikační čísla. Názvy těchto jedinců tedy v tomto případě obsahují typ vazby a pořadové číslo. Ukázkovým příkladem může být vazba typu "owner". Jestliže se jedná o první výskyt tohoto typu vazby v modelu, pak jedinec bude mít název owner1. Pochopitelně, jestliže se v modelu vyskytuje více vazeb stejného typu, pořadové číslo se bude zvyšovat, aby každý jedinec měl svůj unikátní název.

Na následujícím kódu je znázorněna metoda `setIdConnection()`. V prvním kroku je zjišťována existence již uložených vazeb stejného typu a tedy i nastavení pořadí vytvářené vazby. Na dalších řádcích je již klasicky vytvořen jedinec a přiřazen k příslušné třídě.

```
1     public void setIdConnection() {
2        individuals = localBPStudio.getIndividualsInSignature();
3        int i = 1;
4        for(OWLIndividual individual : individuals) {
5           if (individual.toString().contains(type)) {
6              i++;
7           }
8        }
9        for(OWLClass cl : classes) {
10          if (cl.toString().contains(type)) {
11             individual = dataFactory.getOWLNamedIndividual(":" + type + i, pm);
12             classAssertion = dataFactory.getOWLClassAssertionAxiom(cl, individual);
13          }
14       }
15       manager.addAxiom(localBPStudio, classAssertion);
16    }
```

Další metody jako jsou `setName(String name)`, `setDesc(String desc)`, `setExternality(boolean ext)`, `setCoordinateX(int x)`, `setCoordinateY(int y)`, `setMultiplicityFrom(String multiplicity)` a `setMultiplicityTo(String multiplicity)` nastavují další potřebné parametry jednotlivým jedincům. Vždy je použit již vytvořený jedinec a je k němu přiřazena daná vlastnost pomocí `OWLDataPropertyAssertionAxiom`. Tyto axiomy jsou poté pomocí managera přidávány do pracovní ontologie.

V mnoha případech je nutné svázat jednotlivé jedince k sobě pomocí objektových vlastností, jako je tomu například u jedinců typu `Active/Passive`, `Attribute` a `Service`. K tomuto účelu slouží metody `setAttributeRelation()` a `setServiceRelation()`. Toto však nejsou jediné objektové vlastnosti, které je třeba vytvořit. Dalšími důležitými metodami jsou `setConnectionFrom(long idFrom)` a `setConnectionTo(long idTo)`. Tyto metody

označují jedince, kteří jsou danou vazbou propojeni. Do metod jsou předávány id čísla prvků, podle kterých lze najít dané jedince. Nalezený jedinec je poté pomocí třídy `OWLObjectPropertyAssertionAxiom` přiřazen k dané objektové vlastnosti.

Následující příklad ukazuje použití metody `setConnectionFrom(long idFrom)`. Nejprve je vyhledán jedinec na základě předaného id čísla prvku. Do podmínky je nutné zadat také parametr, který omezuje vyhledávání id čísla s pomlčkou. Jak už totiž bylo dříve řečeno, název jedince `Attribute` nebo `Service` se skládá z "type-order-id". Jestliže by tedy byla podmínka nejednoznačná a vyhledáván by byl jedinec pouze podle id, výsledkem by byly také jedinci typu `Service` nebo `Attribute`, což není žádoucí.

```
1   public void setConnectionFrom(long idFrom) {
2      for(OWLNamedIndividual individ : individuals) {
3         if ((individ.toString().contains(String.valueOf(idFrom))) &&
          (!individ.toString().contains("-"+String.valueOf(idFrom)))) {
4            temp = individ;
5            break;
6         }
7      }
8      OWLObjectPropertyAssertionAxiom objectPropertyAssertion = null;
9      for(OWLObjectProperty objectproperty : objectProperties) {
10        if (objectproperty.toString().contains("hasConnectionFrom")) {
             objectPropertyAssertion =
11           dataFactory.getOWLObjectPropertyAssertionAxiom(objectproperty,
             individual, temp);
12        }
13     }
14     manager.addAxiom(localBPStudio, objectPropertyAssertion);
15  }
```

Metodou, která završí celý proces převodu prvku do ontologie je metoda `createElement()`, ve které se uloží všechny provedené změny nad pracovní ontologií. V poslední fázi je pracovní verze ontologie převedena do ontologie s předem definovanou cestou uložení pomocí metody `save(File file)`.

Součástí metody `save(File file)` je také možnost uložení souboru do zvoleného formátu. OWL API poskytuje celou řadu formátů, jako je například klasické OWL/XML, RDF/XML či formáty méně běžné KRSS2, Turtle, Manchester OWL Syntaxe, Latex nebo OWL Functional Syntaxe. V OWL API existuje pro každý formát samostatná třída. Pro uložení ve zvoleném formátu je potřeba vytvořit instanci daného formátu a poté použít metodu `saveOntology(OWLOntology arg0, OWLOntologyFormat arg1, IRI arg2)`, do které je předána samotná ontologie, zvolený formát a místo uložení. Na následujícím příkladě je znázorněno uložení souboru ve formátu RDF/XML.

```
1   public void save(File file) {
2      try {
```

```
3           OWLOntologyFormat format = manager.getOntologyFormat(localBPStudio);
4           RDFXMLOntologyFormat rdfxmlFormat = new RDFXMLOntologyFormat();
5           if(format.isPrefixOWLOntologyFormat()) {
6              rdfxmlFormat.copyPrefixesFrom(format.asPrefixOWLOntologyFormat());
7           }
8           manager.saveOntology(localBPStudio, rdfxmlFormat,
            IRI.create(file.toURI()));
9           } catch (OWLOntologyStorageException e) {
10             e.printStackTrace();
11          }
12       }
13    }
```

7.4. Import ontologie

Součástí této práce je také implementace načtení modelu uloženého v ontologii. Podmínkou je pochopitelně ontologie dříve vytvořená pomocí BP Studia. Načítání ontologie lze rozdělit do dvou částí. První částí je načtení samotného souboru a extrahování informací. Druhá část obsahuje ukládání dat do struktury BP Studia a vykreslení.

Načtení ontologie je naprogramováno ve třídě `LoaderOntology`. Nejprve je potřeba vytvořit OWL manager. Díky němu lze načíst zadanou ontologii. Z této ontologie se následně načtou jedinci do množiny `Set<OWLNamedIndividual>`. Vše je provedeno pomocí metody `load(File file)`.

Pomocí metody `parse()` se postupně tato množina prochází a získává do jednotlivých proměnných parametry, týkající se daného jedince. Veškeré údaje získané z ontologie jsou zapisovány do struktury `Vector<Structure>`. Existuje vždy jeden vektor pro jeden typ prvku (`Process`, `Active`, `Passive`, `Attribute`, `Service`, `Connection`). Třída `Structure` obsahuje proměnnou id, která představuje jedinečný identifikátor prvku a dále také obsahuje hashovou tabulku `HashMap<String, String>`, která slouží k ukládání vlastností a jejich hodnot.

Nejprve jsou tedy v metodě `parse()` načteny datové vlastnosti. Datové vlastnosti jsou uloženy do množiny `Set<OWLDataPropertyExpression>`. Z této struktury lze jednoduše získat vždy název vlastnosti a její hodnotu. Tyto údaje jsou poté ukládány do hashové tabulky. Stejným způsobem jsou načteny také objektové vlastnosti. V případě prvků vazby je potřeba doplnit i vlastnost `hasTypeFrom/To`, která určuje typ prvku, který je s vazbou spjat. Tyto údaje jsou důležité při výsledném vykreslování.

V uvedeném příkladě je znázorněno získávání objektových vlastností, společně s přidáním vlastnosti `hasTypeFrom/To`. Hodnoty těchto vlastností jsou získávány z názvu jedince a je tedy nutné ještě odstranit identifikátor, aby zůstala pouze řetězcová hodnota (`Process`, `Active` nebo `Passive`).

```
1   for(OWLIndividual indiv : objectPropertyValue) {
2      String elementId = indiv.toStringID().replace(base, "");
3      if(objectPropertyName.equals("hasConnectionFrom")) {
4         map.put("hasTypeFrom",
          Pattern.compile("[0-9]").matcher(elementId).replaceAll(""));
5      } else if(objectPropertyName.equals("hasConnectionTo")) {
6         map.put("hasTypeTo",
          Pattern.compile("[0-9]").matcher(elementId).replaceAll(""));
7      }
8      map.put(objectPropertyName,
       Pattern.compile("[a-zA-Z]").matcher(elementId).replaceAll(""));
9   }
```

Jednotlivé struktury načtených dat jsou poté procházeny pomocí metody `save()` třídy `DirectorImport`, která má za úkol data uložit a vykreslit. Vždy je vytvořena instance dotyčné třídy, do které prvek patří. Dále jsou nastaveny veškeré údaje daného prvku společně s nastavením pozice. Poté následuje nahrání celého seznamu prvků stejného typu do slovníku elementů `ElementDictionary`, které slouží jako úložiště veškerých vytvořených prvků, a také nahrání dat do struktury pro grafické zobrazení. V tomto případě je vždy nutné vytvořit k danému prvku také ještě jeho grafickou verzi, a tedy instanci třídy `Node`.

Následující příklad ukazuje vytváření prvků typu proces. Je vytvořena instance tříd `BusinessProcess`, nastaveny jednotlivé parametry prvku. Na řádku 15 je volána metoda `writeProcess()`, která má za úkol uložit vytvořené instance do struktury BP Studia, jak je vidět na následujících řádcích. Do diagramu jsou poté přidány nové procesy a vše je následně vykresleno.

```
1   for(Structure str : dataProcess) {
2      Map<String,String> map = str.getMap();
3      BusinessProcess process = new BusinessProcess();
4      process.setId(str.getId());
5      process.setName(str.getMap().get("hasName"));
6      process.setSpec(str.getMap().get("hasDesc"));
7      process.setExternal(Boolean.getBoolean(str.getMap().get("hasExt")));
8      Point point = new Point();
9      x = Integer.valueOf(str.getMap().get("hasCoordinateX").toString());
10     y = Integer.valueOf(str.getMap().get("hasCoordinateY").toString());
11     point.setLocation(x, y);
12     points.add(point);
13     processes.addElement(process);
14  }
15  writeProcesses();
16
17  public void writeProcesses() {
18     int i = 0;
19     method.elementDictionaries.setProcesses(processes);
20     Vector<? extends ProcessElement> elements =
       method.getElements(ElementType.PROCESS);
21     for (ProcessElement el : elements) {
22        Vector<Diagram> diagrams = method.getDiagrams(DiagramType.FUNCTIONAL);
23        for (Diagram diagram : diagrams) {
24           Vector<Node> nodes = diagram.getNodes();
25           ProcessNode processNode = new ProcessNode(el);
```

```
26              processNode.setOrigin(points.get(i));
27              nodes.add(processNode);
28          }
29          i++;
30      }
31  }
```

V případě načítání vazeb je opět celý proces o něco složitější. Na základě typu vazby je vybrána metoda `insertConnection(ElementFactory factory)`, která vytváří novou instanci konkrétního typu. V této metodě jsou dále definovány další vlastnosti vazby, jako je například kardinalita, popis a podobně. Při ukládání dat o vazbě je také potřeba načíst grafickou reprezentaci prvků, které vazba spojuje. To je vyřešeno pomocí metody `getNode(String type, long id)`, kdy právě v tomto momentě jsou využity informace vlastnosti `hasTypeFrom/To`. Díky této informaci společně ještě s jednoznačným identifikátorem lze totiž jednoduše získat odkaz na daný prvek.

8. Builder

8.1. Popis

Návrhový vzor je obecná šablona řešení problému. Návrhové vzory se využívají při návrhu programu. Jedná se o doporučený postup řešení, není tedy nutné přesně dodržovat veškerá pravidla s nimi spjatá, ale je možné využít pouze zásadní myšlenky. Díky návrhovým vzorům se výrazně sníží pravděpodobnost chyb, které by bylo možné udělat během implementace, a také se ulehčí práce při doplňujícím rozšiřování kódu, jelikož vzory právě s tímto počítají.

Návrhové vzory jsou tvořeny strukturou tříd, které využívají typické vlastnosti pro objektově orientovaného programování. Díky návrhovým vzorům je také celý kód mnohem čitelnější a srozumitelnější i pro někoho, kdo tento kód nevyvíjel [9].

8.2. Builder

V této diplomové práci je využit návrhový vzor Builder neboli Stavitel. Tento návrhový vzor slouží pro vytváření objektů. Jeho hlavním účelem je oddělit konstrukci daného objektu od jeho vnitřní reprezentace. Tím pádem umožňuje také využít stejného konstrukčního postupu pro různé typy objektů.

Tento návrhový vzor se skládá z několika základních prvků. Je jím samotný builder, který zastává roli výkonného objektu. Jeho nadřazeným prvkem je tzv. řídící prvek - director. Director ví, jak se má daný objekt vytvořit. Dostane přidělený builder a poté postupně přiděluje úkoly, které builder vykonává. Builderů může být samozřejmě více. Každý pak představuje určitý typ zpracování, co je potřeba pro vytvoření konkrétního objektu udělat. Cílem tohoto vzoru je umožnit definovat nové výkonné objekty, aniž by bylo nutné měnit kód řídícího objektu. Využívá se tehdy, jestliže se vytvářené objekty budují postupně v několika fázích. Director v tomto případě pak dohlíží na správnou posloupnost provádění jednotlivých fází objektu. Dalším využitím může být například vytváření objektů, při kterých jsou využívány systémové zdroje, které jsou omezené. Director v tomto případě řídí celý proces a přiděluje jednotlivé zdroje podle potřeby.

V případě této diplomové práce se návrhový vzor Builder využívá pro potřeby ukládání modelu do formátu ontologie. Umožňuje však také dodatečnou implementaci i jiných formátů, což činí kód pro export univerzální. Navíc je celý kód čitelnější a srozumitelnější. Jelikož BP Studio podporuje také i import souborů, existuje jak struktura tříd pro exportování, tak také pro importování souboru.

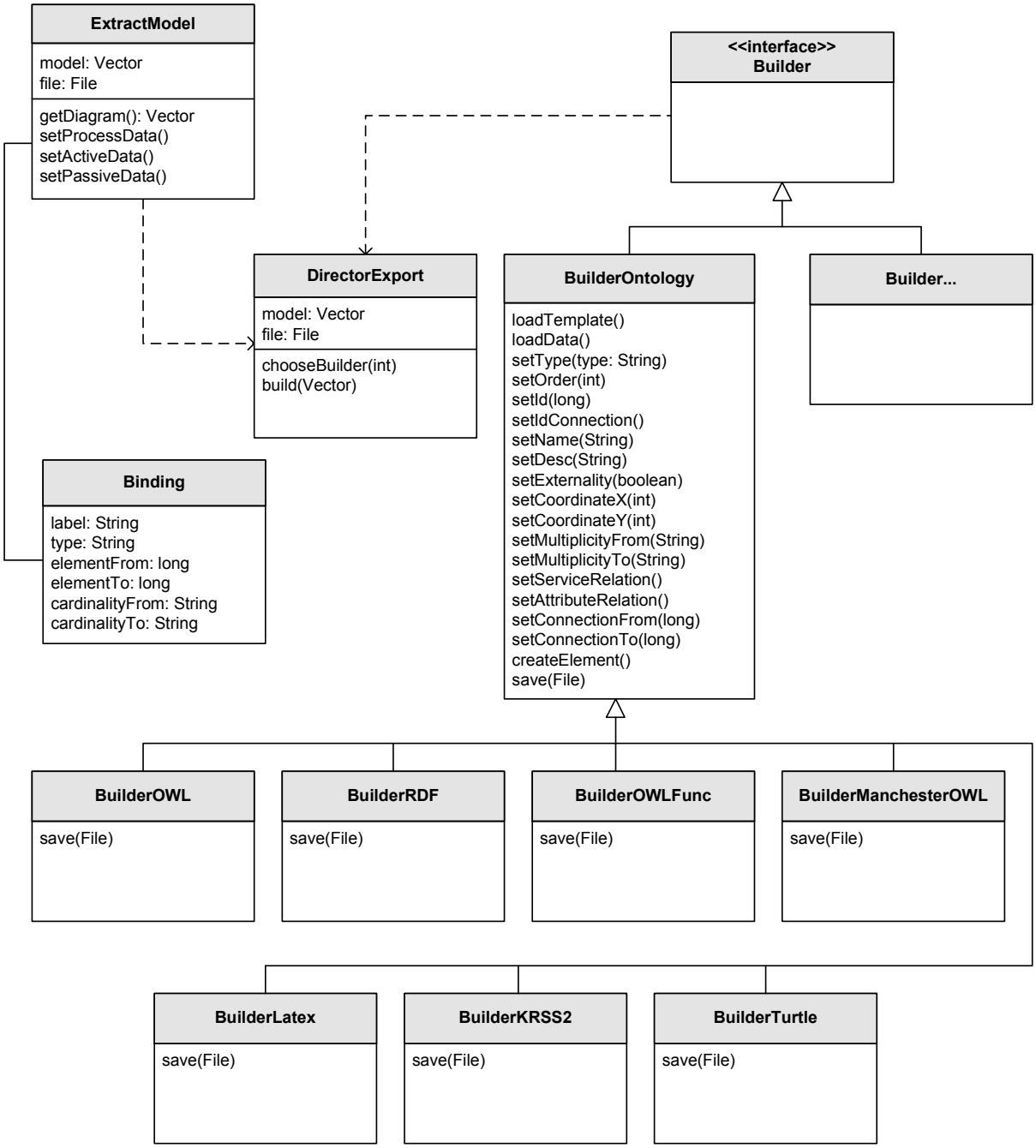
8.3. Export

Export modelu z BP Studia je implementován v balíčku `bpm.builder`. Celá cesta exportování modelu začíná při otevření hlavního menu `Soubor -> Export`. V nabídce je velké množství různých ontologických formátů (OWL/XML, RDF/XML, OWL Functional Syntax, Manchester OWL Syntax, Latex, KRSS2 Syntax a Turtle). Při výběru jednoho z nich se zobrazí save-dialogové okno, ve kterém uživatel nadefinuje cestu ukládaného souboru. Při potvrzení tohoto okna je řízení předáno třídě `DirectorExport`. Tato třída zastává roli řídícího prvku. Nejprve jsou pomocí třídy `ExtractModel` získána data namodelovaného diagramu. Poté následuje výběr builderu, který bude s daty dále pracovat. Výběr je závislý na zvoleném formátu ukládaného souboru a provádí se pomocí metody `chooseBuilder(int format)`.

V metodě `build(Vector model)` jsou postupně procházena data v závislosti na typu prvku. Pro každý typ prvku jsou poté volány jednotlivé metody builderu pro zápis do souboru. U každého prvku je nejprve nutné nastavit jeho typ a poté identifikátor. Další parametry jsou voleny podle konkrétního typu prvku. U prvků typu `Active` a `Passive` je nutné vytvořit nový typ prvku (`Attribute` nebo `Service`) a také mu nastavit potřebné parametry. Uložení souboru je také součástí builderu a je provedeno pomocí volání metody `save(File file)`. Tato metoda je překryta metodou v závislosti na zvoleném formátu. V metodě je předán soubor a tedy i finální umístění, které bylo zvoleno na počátku.

Součástí builderu pro vytváření ontologií jsou také třídy přepisující metodu `save(File file)` pro konkrétní formát souboru. Existuje tedy například třída `BuilderOWL`, která uloží ontologii ve formátu OWL/XML či třída `BuilderRDF`, která vytváří ontologii RDF/XML.

Na následujícím třídním diagramu je znázorněna celá struktura builderu. V tomto případě je vytvořen pouze builder pro zápis ontologických souborů s název `BuilderOntology`. Je zde uvedena i třída `Binding`, která slouží jako pomocná strukturu pro uložení prvků vazby.



Obrázek 28: Struktura tříd pro export

8.4. Import

Import modelu ze souboru se provádí spuštěním nabídky `Soubor -> Import -> Ontology`. Zobrazí se open-dialogové okno, ve kterém uživatel definuje pozici souboru. Následně se řízení předá třídě `DirectorImport`, která je společně i s ostatními použitými třídami uložena v balíčku `bpm.loader`. `DirectorImport` má za úkol řídit celý proces otevření a načtení modelu do

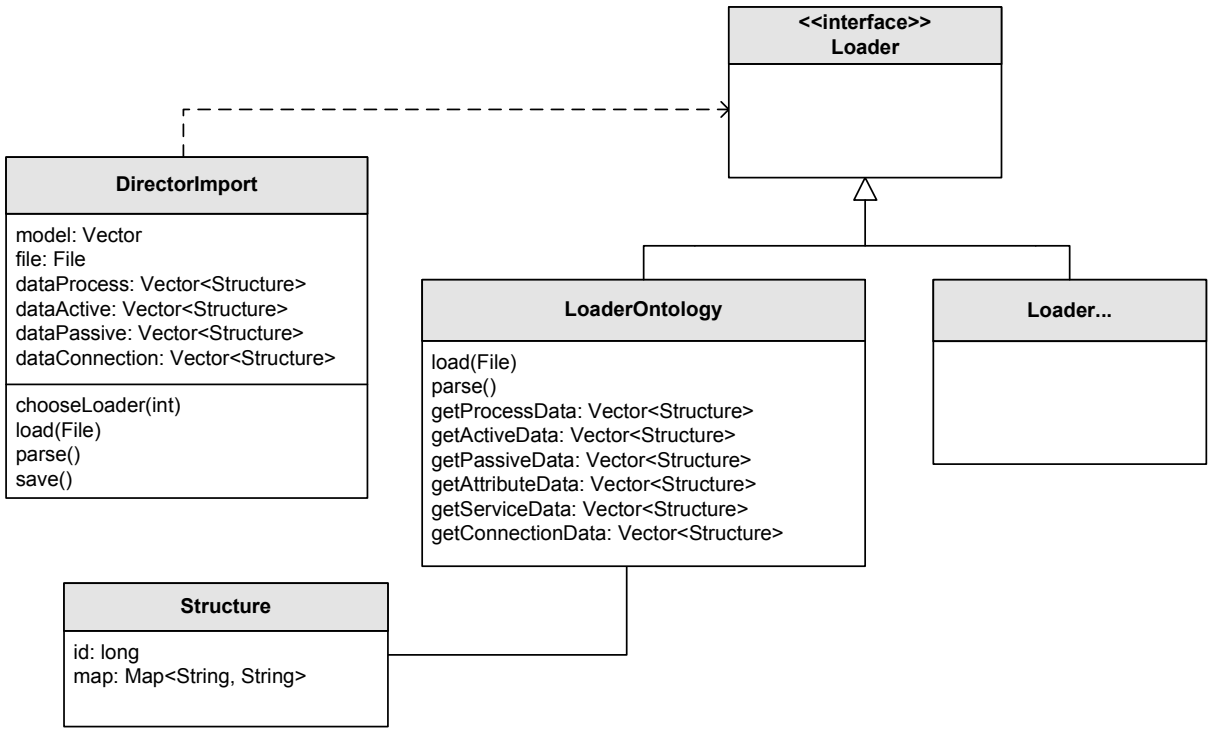
BP Studia. Nejprve je na základě souborového formátu vybrána konkrétní třída pro načítání souboru, tzv. loader. O to se stará metoda `chooseLoader(int format)`.

Následuje volání metody `load(File file)`. Tato metoda se nachází v loaderu a slouží k načtení souboru do paměti. Pomocí metody `parse()` dochází k extrakci dat ze souboru a rozdělení do jednotlivých struktur typu `Vector<Structure>` podle typu prvku.

Metoda `save` třídy `DirectorImport` ukládá data do struktury aplikace BP Studio a následně je vykresluje pomocí vyplnění struktury pro grafické zobrazení prvků.

V rámci exportu je možné diagram ukládat ve více různých ontologických formátech. Při zpětném importu však nehraje roli o jaký formát se jedná, tudíž není potřeba vytvářet zvláštní třídu pro každý formát.

Na následujícím diagramu lze vidět strukturu celého mechanismu pro import. V uvedeném příkladě je znázorněn pouze loader pro načítání ontologických souborů s názvem `LoaderOntology`. Je zde uvedena i třída `Structure`, která slouží jako pomocná struktura pro uložení dat získaných z ontologie.



Obrázek 29: Struktura tříd pro import

8.5. Přidání nového formátu

Celá konstrukce exportu a importu je navržena pomocí návrhového vzoru Builder a tudíž je určena mimo jiné také pro dodatečné přidávání funkcionality týkající se převodu na další formát. Struktura

umožňuje jednoduché přidání další třídy, aniž by bylo potřeba zásadně měnit a zasahovat do kódu programu.

V první řadě je potřeba vytvořit novou položku menu představující další formát. Tato definice se vytváří ve třídě `BPModel` v balíčku `bpm.gui.model`. V dalším kroku je potřeba v řídící třídě (`DirectorExport`/`DirectorImport`) v metodě `chooseBuilder(int format)`/`chooseLoader(int format)` přidat výběr dalšího formátu. Na následujícím příkladě je zobrazena metoda pro výběr správného builderu. Na výběr jsou různé ontologické formáty.

```
1   public void chooseBuilder(int format) {
2      switch(format) {
3      case 1: { builder = new BuilderOWL(file); break;}
4      case 2: { builder = new BuilderRDF(file); break;}
5      case 3: { builder = new BuilderOWLFunc(file); break;}
6      case 4: { builder = new BuilderManchesterOWL(file); break;}
7      case 5: { builder = new BuilderLatex(file); break;}
8      case 6: { builder = new BuilderKRSS2(file); break;}
9      case 7: { builder = new BuilderTurtle(file); break;}
10     }
11  }
```

Poté již stačí pouze vytvořit třídu obsluhující vytváření souboru (`Builder...`) a třídu pro načtení a parserování souboru (`Loader...`). Další úpravy již nejsou nutné, jelikož se o vše postará řídící třída. Třída pro export diagramu do souboru by měla implementovat rozhraní `Builder` a obsahovat tak metody pro načtení šablony a vytvoření jednotlivých prvků. Třída starající se o import souboru by měla implementovat rozhraní `Loader` a měla by mít metody pro načtení souboru a jeho parserování.

9. Závěr

Ve světě moderní informatiky zaujímají ontologie významné místo. Pomocí ontologií lidé dokáží modelovat realitu z mnoha různých úhlů. Ontologie se již v dnešní době využívá pro rozšíření databázových schémat, v oboru umělé inteligence například pro samostatné odvozování či při vytváření webových metadat. Ontologie však není omezena pouze na tyto obory, ale je velice vhodná také při práci v oblasti softwarového inženýrství.

Tato práce se zabývá transformací modelů business procesů do ontologií. Klasická formální ontologie však není vždy pro člověka snadno čitelná. Proto jedním z řešení je grafický nástroj, který je ontologiemi podporován na pozadí. Tímto nástrojem je editor BP Studio, ve kterém je díky této práci implementován nový modul pro konverzi modelu do ontologie a zpět. Prostřednictvím tohoto propojení je možné s diagramy rozmanitěji pracovat a využívat sémantického významu ontologií.

Obsahem této práce byla transformace pouze funkčního diagramu. V dalších krocích by bylo samozřejmě vhodné do transformace zapojit také diagram v objektovém a koordinačním pohledu. V tomto případě by nebylo potřeba vytvářet novou ontologii, pouze by byla doplněna o další třídy a vlastnosti. Struktura implementace je navržena také tak, aby bylo možné dodatečně pohodlně přidávat i moduly pro transformaci do jiných formátů.

Z této diplomové práce vznikl také odborný článek, který bude přednesen na konferenci EDOC 10.-14. září 2012 [10].

Literatura

[1] TANIAR, David a Johanna Wenny RAHAYU. *Web semantics and ontology*. Hershey: Idea Group Publishing, 2006. ISBN 1-59140-905-5.

[2] GRUBER, T.R.: *A Translation Approach to Portable Ontology Specifications*. Knowledge Acquisition, 5(2) (1993).

[3] BORST, W.N.: *Construction of Engineering Ontologies for Knowledge Sharing and Reuse*. PhD dissertation, University of Twente, Enschede, 1997.

[4] SVÁTEK, V. *Ontologie a WWW*. VŠE DATAKON 2002, Brno. 2002. Dostupné z: http://nb.vse.cz/~svatek/onto-www.pdf

[5] ROGALEWICZOVÁ, Tereza. *Ontologie a jejich využití v geoinformatice*. Praha, 2008. Diplomová práce. ČVUT.

[6] HORRIDGE, Matthew. PROTÉGÉ. *A Practical Guide To Building OWL Ontologies Using Protégé 4 and CO-ODE Tools* [online]. 1.3. The University Of Manchester, 24.3.2011 [cit. 2012-04-16]. Dostupné z: http://owl.cs.manchester.ac.uk/tutorials/protegeowltutorial/resources/ProtegeOWLTutorialP4_v1_3.pdf

[7] HORRIDGE, Matthew a Sean BECHHOFER. *The OWL API: A Java API for Working with OWL 2 Ontologies*. OWLED 2009, 6th OWL Experienced and Directions Workshop, Chantilly, Virginia, October 2009. Dostupné z: http://www.webont.org/owled/2009/papers/owled2009_submission_29.pdf

[8] VONDRÁK, I.: *Business Process Modeling and Simulation for Quality Management*. In European Simulation Multiconference ESM 2000. Ghent, Belgium: . 2000. 375-379. SCS. 1-56555-204-0

[9] PECINOVSKÝ, Rudolf. *Návrhové vzory*. Brno: Computer Press, 2007. ISBN 978-80-251-1582-4.

[10] ŠTOLFA, Jakub, Svatopluk ŠTOLFA, Jan KOŽUSZNIK a Tereza MOUDRÁ. *Business Process Formal Modeling in Graphical Ontology Tool*: *Functional View*. Ostrava, 2012. Odborný článek. VŠB - Technická univerzita Ostrava.

Seznam příloh na CD

Adresář	Obsah adresáře
/src	Zdrojové kódy diplomové práce
/template	Ontologie
/text	Textová část diplomové práce
/img	Obrázky